OCaml

Programação funcional na prática

Casa do Código

ANDREI DE ARAÚJO FORMIGA

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

ISBN

Impresso e PDF: 978-85-5519-070-4

EPUB: 978-85-5519-071-1

MOBI: 978-85-5519-072-8

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

PREFÁCIO

Este é um livro sobre a linguagem de programação OCaml e sobre o paradigma de programação funcional. Não faz tanto tempo, o autor de um livro como este teria de gastar um número muito maior de palavras justificando para o leitor por que ele deve empreender um esforço para sair da sua zona de conforto e aprender um novo paradigma.

Hoje em dia, essa tarefa é muito mais fácil, quando consideramos as tendências atuais no universo do desenvolvimento de *software*. Cada vez mais, a indústria de software se convence das vantagens do paradigma funcional, e muitas das novas e promissoras ideias sobre programação que apareceram nos últimos anos vieram da comunidade da programação funcional.

A Orientação a Objetos (OO) ainda é dominante e continua sendo amplamente usada, mas parece um pouco estagnada: poucas novas ideias parecem estar surgindo desse paradigma. A maior parte das novidades nas novas versões de linguagens populares, como Java, C# e C++, tem inspiração funcional.

Um exemplo são os *lambdas* (funções anônimas que geram *closures*) adicionados na versão 8 do Java e no padrão C++11. A cada nova versão, o JavaScript ganha mais e mais características funcionais. Os dois principais ecossistemas para aplicações gerenciadas, as plataformas Java e .NET, ambos possuem uma ou mais linguagens primariamente funcionais de crescente popularidade; Scala e Closure no caso da plataforma Java, e F# em .NET. F# começou basicamente como uma versão de OCaml para a plataforma .NET.

Novas linguagens como a Rust, criada pela Mozilla, a Swift,

criada pela Apple, e a Hack, criada pelo Facebook, carregam também uma grande influência funcional. Por sinal, o compilador de Hack é escrito em OCaml. Para programadores que desejam se manter atualizados e em contato com as boas práticas da indústria, fica cada vez mais difícil evitar a programação funcional.

Entretanto, embora já exista um bom número de livros sobre linguagens funcionais disponíveis em inglês, tanto livros mais acadêmicos quanto mais práticos, as opções na língua portuguesa ainda são escassas. Este livro procura preencher essa lacuna, mas de forma menos acadêmica e mais pragmática.

A forma mais prática de aprender um paradigma é se aprofundando em uma linguagem importante desse paradigma. Daí este livro, OCaml: programação funcional na prática.

Por que "na prática"?

Muitos livros sobre programação funcional são livros-textos criados para serem material didático em disciplinas sobre programação funcional ou paradigmas de linguagens de programação. Isso não é um problema em si; muitos desses livros-textos são excelentes e valem a pena serem lidos. Entretanto, eles frequentemente incluem tópicos que, embora sejam interessantes, não são essenciais para quem quer programar em uma linguagem funcional.

Um exemplo é falar sobre o lambda-cálculo, que é o fundamento teórico das linguagens funcionais; é um tópico interessante, mas não necessariamente essencial para programadores iniciantes no modelo. Por outro lado, existem características de OCaml que não são tradicionais nos cursos sobre programação funcional, mas que são frequentemente usadas na prática pelos programadores; dois exemplos são os parâmetros rotulados e as variantes polimórficas.

Este livro é prático porque foi escrito pensando no leitor como um programador interessado em OCaml e no paradigma funcional, não necessariamente um aluno de algum curso. A intenção é equipar o leitor para ler código escrito pela comunidade da linguagem e para criar projetos de seu interesse em OCaml, além de mostrar seus princípios fundamentais.

Livros-textos tendem a um enfoque maior nos princípios e na teoria, e menor nos detalhes de linguagens específicas. Como um grande tópico, a programação funcional é maior do que apenas OCaml (e OCaml não é apenas funcional), mas consideramos que a melhor forma de aprender um paradigma mais a fundo é se aprofundar em uma linguagem específica desse paradigma. Essa é a intenção deste livro.

Desta forma, o livro inclui várias dicas e sugestões de como usar OCaml de maneira mais efetiva, além de observações sobre o que é e não é idiomático na linguagem, de acordo com o código que é produzido pela comunidade de usuários. Embora o livro não cubra algumas características mais recentes e avançadas, a intenção é de que quem leia o livro inteiro tenha condições de entender 90% ou mais do código escrito em OCaml que pode ser encontrado na internet.

Os tópicos cobertos no livro foram escolhidos com essa intenção. Com essa base coberta, os leitores mais interessados poderão aprender os aspectos mais profundos da linguagem nos manuais ou em outros materiais.

Este livro também é prático porque inclui, além de vários exemplos pequenos que ilustram características específicas, alguns exemplos maiores de programas relativamente realistas em OCaml. Esses exemplos demonstram, ao mesmo tempo, alguns domínios em que a programação funcional se sai bem, e áreas de aplicação que são interessantes para um programador.

O código-fonte desses exemplos é criado e organizado de uma maneira comum para projetos em OCaml, ilustrando como algumas ferramentas presentes no ecossistema da linguagem (por exemplo, ferramentas de *build* e de teste) são usadas em projetos reais. Dessa forma, o leitor pode usar os exemplos mais longos como modelos (em termos de organização e ferramentas utilizadas) para seus próprios projetos.

O que se espera do leitor e o que o leitor pode esperar do livro

O presente livro foi escrito pensando no leitor como um programador com alguma experiência em linguagens imperativas/OO, mas que não teve contato anterior com o paradigma funcional, ou teve um contato apenas superficial. Esse ponto de referência é utilizado para centrar as explicações ao longo do texto.

OCaml não é uma linguagem minimalista nem tem a pretensão de ser "simples"; é expressiva e poderosa com um conjunto considerável de *features* adicionadas por cima do núcleo funcional da família ML. A sequência de capítulos do livro pretende introduzir a maior parte dessas *features* em uma ordem que faça sentido para um iniciante na linguagem.

Ao fim do capítulo *Programação funcional*, o leitor terá visto todo o núcleo funcional; o fim do capítulo *Características imperativas* marca a exposição completa do que podemos chamar de **núcleo básico** das linguagens ML. Exceto por diferenças sintáticas, esse núcleo é praticamente o mesmo em OCaml, Standard ML e F#, três linguagens da família ML.

A partir daí, passamos a cobrir características mais avançadas de OCaml e que a tornam mais única dentro de sua família. No meio dos capítulos introduzindo novas partes da linguagem, temos dois

com estudos de caso, mostrando programas mais substanciais em OCaml: um dos capítulos mostra um compilador, um interpretador e uma máquina virtual para linguagens simples; o outro mostra como analisar dados e como empregar técnicas de aprendizado de máquina para classificar informações automaticamente.

A forma recomendada de ler o livro, como qualquer outro livro de programação, é ativamente: tentando os exemplos, fazendo perguntas ao texto (e tentando respondê-las com experimentação), usando o que foi introduzido para escrever alguns programas pequenos mas interessantes etc. Como disse Aristóteles, aquilo que devemos aprender a fazer nós aprendemos fazendo.

O site do livro inclui como material adicional, algumas sugestões de exercícios e projetos de vários níveis de esforço e complexidade para um leitor interessado em praticar mais.

Site e material adicional

Acompanhando o livro, existe um site com material adicional sobre a linguagem OCaml, incluindo exercícios, maneiras de fazer perguntas ou discutir sobre o livro, textos sobre assuntos não cobertos no texto, indicações de projetos para escrever em OCaml, e potencialmente mais.

É importante para quem quer aprender mais sobre a linguagem ou o paradigma funcional não apenas ler o livro, mas se engajar com o assunto. O endereço é http://andreiformiga.com/livro/ocaml.

AGRADECIMENTOS

Escrever um livro é uma tarefa cansativa, mas recompensadora, e que se torna possível apenas com a ajuda de algumas pessoas. Embora muita gente adicione algo à soma de experiências pessoais que compõem a vida do autor, eu gostaria de agradecer em especial a algumas delas.

Primeiramente a meus pais, pelo fato clichê, mas verdadeiro, de que eles sempre fizeram tudo ao alcance para que eu tivesse acesso a uma educação de qualidade e ao conhecimento. Nunca negaram um pedido meu para comprar livros — e olhe que eu pedi muitos.

À Casa do Código e ao Paulo Silveira, por terem embarcado na ideia de fazer um livro em português sobre uma linguagem pouco conhecida, que nem é a linguagem funcional mais famosa. Espero que este livro contribua para que ela receba um maior reconhecimento, merecidamente.

A Gio e João, pela convivência familiar agradável (na maior parte do tempo) e recompensadora, pelo apoio incessante e por tomar a frente em algumas tarefas em que eu não podia contribuir mais, pois estava escrevendo. José Antônio ainda está em compilação (em um *build* de nove meses), mas já serve como fonte de inspiração.

Aos amigos, Maurício Linhares e Yuri Malheiros, por terem lido alguns capítulos e sugerido ótimas ideias, algumas das quais foram implementadas e tornaram o livro melhor.

Ao leitor que decidiu pegar este livro e aprender algo novo. Apesar do trabalho envolvido, posso dizer com sinceridade que foi divertido escrevê-lo e espero que você se divirta lendo.

SOBRE O AUTOR

Andrei de Araújo Formiga é professor no Centro de Informática da Universidade Federal da Paraíba, mas jura que ainda sabe programar. Apaixonou-se pela programação ainda cedo e nunca a deixou, embora às vezes seja um relacionamento à distância por conta de outras obrigações. Mas o reencontro é sempre emocionante.

Suas áreas de interesse como professor, pesquisador e eventual *hacker* da Torre de Marfim incluem tudo sobre linguagens de programação (teoria, semântica, implementação etc.). Recentemente, tem se interessado muito por aprendizado de máquina e temas relacionados, mas também se diverte lendo livros de matemática e textos escritos por Knuth.

Seu site está localizado em http://andreiformiga.com.

Sumário

Capítulo 1

INTRODUÇÃO

OCaml é uma linguagem funcional com tipagem estática criada pelo instituto de pesquisa francês INRIA (*Institut National de Recherche en Informatique et en Automatique*). O "Caml" no nome é pronunciado como "camel" mesmo, e por isso os logotipos da linguagem geralmente representam um camelo (veja figura a seguir). Fora isso, não é preciso se preocupar muito em como pronunciar o nome da linguagem, já que os falantes de língua inglesa pronunciam de uma forma e os criadores da linguagem, que são franceses, pronunciam de outra.

Figura 1.1: Logotipo da linguagem OCaml

1.1 POR QUE PROGRAMAÇÃO FUNCIONAL?

Aprender um novo paradigma de programação requer sair da zona de conforto. Os primeiros passos são difíceis e, às vezes, é

frustrante tentar se expressar e resolver problemas no novo modelo, pensando que, se fosse uma linguagem que conhecemos, o mesmo problema seria resolvido em poucos minutos.

Mas em compensação, aprender um novo paradigma nos dá uma nova visão sobre programação e como resolver problemas. Essa nova perspectiva rende benefícios que vão além de linguagens de programação específicas, e gera *insights* que podem ser usados em muitas linguagens diferentes, mesmo que não sejam do novo paradigma.

"Uma linguagem que não afeta sua forma de pensar sobre programação é uma linguagem que não vale a pena conhecer" — Alan Perlis

Se você está lendo até aqui, é porque já tem algum interesse em aprender programação funcional. Talvez você tenha visto que várias linguagens populares estão incorporando mais características funcionais em cada versão.

Talvez você tenha percebido que várias das novas linguagens que estão sendo discutidas por desenvolvedores no mundo todo, como Rust, da Mozilla, Hack do Facebook ou Swift da Apple, são fortemente influenciadas pela programação funcional. Talvez você tenha ligado os pontos e visto que a programação funcional se torna cada vez mais *mainstream*, e conhecer esse paradigma se torna importante até para usar bem linguagens tradicionais, como Java.

Como o objetivo é aprender programação funcional através de uma linguagem, é interessante escolher uma que seja primariamente funcional, mesmo que tenha características de outros modelos. OCaml tem essa característica e é uma boa escolha para aprender o paradigma. Mas não é a única: muita gente já ouviu falar de linguagens como Scala, Clojure, Haskell e F#.

Não que OCaml seja a melhor opção absoluta entre essas, mas ela é uma opção muito boa. Antes de justificar essa afirmação, veremos algumas características da linguagem.

O NOME OCAML REFLETE SUA HISTÓRIA

O nome da linguagem carrega vários aspectos de sua história. OCaml faz parte da família ML de linguagens, que começou com a ML original criada na universidade de Edimburgo por Robin Milner, no começo da década de 70. ML significa *Meta Language*, pois ela foi feita para funcionar como a metalinguagem do provador de teoremas LCF. Mas os primeiros usuários de ML perceberam que ela funcionava bem para programar outras coisas, então ela foi separada do seu sistema de origem.

Com o aumento de popularidade, ainda no meio acadêmico, várias versões de ML foram desenvolvidas em universidades e institutos da Europa. No INRIA da França, Pierre-Louis Curien criou a *Categorical Abstract Machine* (CAM), uma máquina virtual e modelo de computação adequado para execução de linguagens funcionais. Gérard Huet e outros pesquisadores criaram uma versão de ML usando a máquina CAM, e a chamaram de CAML.

A máquina CAM e a linguagem CAML surgiram em meados da década 80, mas a implementação de CAML era considerada "pesada", pois o código executado era lento e usava muita memória. Isso levou Xavier Leroy e Damien Doligez a criarem, no início dos anos 90, a implementação *Caml Light* para a linguagem, que era muito mais rápida e compacta.

Orientação a Objetos estava na moda na década de 90, de modo que a linguagem foi estendida com a adição de classes e

objetos, resultando na *Objective Caml*, que é basicamente a versão usada até hoje. Em 2011, o nome foi mudado oficialmente para *OCaml*, que era o apelido informal usado pela comunidade. Mais detalhes sobre sua história podem ser encontrados em http://caml.inria.fr/about/history.en.html.

1.2 CARACTERÍSTICAS DE OCAML

Acho que a essa altura já deu para perceber que OCaml é uma linguagem funcional (tem até no título do livro). Mas nem toda linguagem funcional é igual, então começamos listando as suas principais características.

Estaticamente tipada: assim como Java, C#, Haskell e Scala, OCaml é estaticamente tipada; o compilador verifica os tipos dos valores e variáveis usados, e não compila código que apresente erros de tipo. Tipos são importantes na programação OCaml não só para detectar erros no código, mas também como forma de expressar ideias relacionadas ao projeto do programa. Esse tema será explorado repetidamente, ao longo do livro.

Inferência de tipos: se mostrarmos um trecho de código OCaml para um programador que não conhece a linguagem (nem outras similares a ela), ele pode achar que é dinamicamente tipada. Por exemplo, segue uma função em OCaml que retorna a diferença entre dois números, em valor absoluto:

```
let dif a b =
  if a > b then a - b else b - a
```

A função se chama `dif`, mas onde estão os tipos? Diferente de uma linguagem como Java, em OCaml o programador geralmente não precisa declarar os tipos de variáveis e funções. O compilador de OCaml determina automaticamente os tipos envolvidos. Para

essa função, é determinado o seguinte tipo:

```
val dif : int -> int -> int
```

Isso significa que a função `dif` recebe dois parâmetros inteiros e retorna um inteiro, como aprenderemos mais tarde. A inferência de tipos significa que código OCaml pode ser compacto como o de uma linguagem com tipagem dinâmica, mas com todas as garantias e vantagens de ter um sistema de tipos com tipagem estática.

Orientada a expressões: diferente da maioria das linguagens imperativas (mas similar a outras funcionais), OCaml é orientada a expressões e não orientada a comandos. Isso significa que todas as estruturas em código OCaml são declarações ou expressões. O interessante de expressões é que elas têm um valor, e as partes de uma expressão que também são expressões (chamadas de *subexpressões*) também têm um valor associado a elas.

A ideia da programação funcional é que o programa seja uma composição de expressões, como ocorre na matemática, em vez de uma lista de comandos como acontece na programação imperativa. As consequências práticas disso serão discutidas a partir do próximo capítulo.

Compilação nativa ou para bytecode: OCaml tem dois compiladores: um que gera código nativo para um processador específico e um que gera código *bytecode* para uma máquina virtual. O compilador para código nativo é o `ocamlopt`, enquanto que o de *bytecode* é `ocamlc`. Ambos devem estar presentes em qualquer instalação do sistema OCaml.

O código compilado nativamente executará mais rápido e é normalmente usado em produção. Em teoria, o código *bytecode* deveria ser portável para diferentes plataformas (como acontece em Java), mas, na prática, não é. A grande vantagem de compilar para *bytecode* é a velocidade de compilação, e a possibilidade de usar o

código compilado para *bytecode* no modo interativo da linguagem (como veremos a seguir). Em geral, usa-se o compilador *bytecode* durante o desenvolvimento, e o compilador nativo em produção.

Uso interativo ou em batch: não é raro ver as linguagens de programação separadas em duas classes: *linguagens compiladas* e *linguagens interpretadas*. Essa classificação segue o senso comum, mas não é muito correta.

Uma linguagem de programação pode ser implementada de várias formas, as mais comuns sendo através de interpretação ou compilação (mesmo a diferença real entre interpretação e compilação seja mais complicada do que a maioria dos programadores tem conhecimento).

OCaml tem, de certa forma, algo parecido com os dois modos: interpretação e compilação. Código OCaml pode ser compilado com um dos compiladores e executado (modo *batch*), ou pode ser usado interativamente, executando código à medida em que é digitado.

Mas esse modo interativo não é, realmente, um interpretador: ele apenas compila cada expressão que é digitada para *bytecode* imediatamente, e o executa quando necessário. O que muitos chamam de interpretador é mais precisamente chamado de *loop interativo* (ou *REPL*, sigla de *Read-Eval-Print Loop*). Porque o REPL faz exatamente isso: é um *loop* contínuo de ler (*Read*) a entrada, executá-la (*Eval*) de alguma forma, e imprimir (*Print*) o resultado. Daqui para a frente, usaremos o termo REPL neste livro (alguns textos que falam sobre OCaml usam o termo *toploop* para o REPL).

ADTs e Pattern matching: assim como acontece em outras linguagens funcionais, como Scala e Haskell, é difícil ler um programa não trivial em OCaml que não use os tipos de dados algébricos (ADTs, do inglês *algebraic data types*) e *pattern matching*.

Os ADTs nos permitem construir valores com estrutura complexa de maneira simples, e o *pattern matching* nos dá a capacidade de examinar valores com estrutura complexa de forma simples. Quem se acostuma a usar essas características por um tempo geralmente não quer mais voltar a programar sem elas.

Módulos: OCaml possui um avançado sistema de módulos para organização do código. Programas simples geralmente existem em um único módulo, mas a capacidade de criar e combinar módulos com características avançadas ajuda na estruturação de aplicações e bibliotecas complexas. Os *módulos de primeira classe* também podem ajudar na organização de uma aplicação em um núcleo fixo e um conjunto de módulos dinamicamente configurados, como um sistema de *plugins*.

Objetos e classes: embora não seja muito usado pela comunidade, OCaml inclui um sistema de objetos que, embora tenha características um pouco diferentes das linguagens orientadas a objetos mais tradicionais, inclui a possibilidade de usar várias práticas do paradigma OO, como herança, encapsulação, métodos virtuais etc.

Avaliação estrita: similar à maioria das linguagens de programação mais conhecidas, mas diferente de linguagens como Haskell, a avaliação em OCaml é estrita. De forma simples, a maneira mais direta de ilustrar a diferença entre avaliação estrita e não-estrita é usando a operação de chamada de função.

Na programação funcional, essa operação é normalmente chamada de *aplicação* de uma função aos seus argumentos. Em uma linguagem estrita, primeiro é determinado o valor de todos os argumentos para a função, e os valores determinados são passados para a função. Em uma não-estrita, esse valor é determinado quando necessário, o que às vezes é chamado de *avaliação preguiçosa*, embora os dois termos (avaliação não-estrita e avaliação

preguiçosa) não signifiquem exatamente a mesma coisa.

O código compilado para uma linguagem usando avaliação estrita tende a ser mais eficiente, e sua eficiência tende a ser mais previsível, principalmente com relação ao uso da memória. Mas a avaliação não-estrita também tem suas vantagens. Um exemplo é que é muito mais simples trabalhar com estruturas de dados infinitas na avaliação não-estrita.

1.3 POR QUE OCAML?

Além do benefício de aprender um novo paradigma de programação, a linguagem OCaml pode ser utilizada com vantagem em projetos de programação. Em comparação com outras linguagens, ela oferece uma boa combinação de poder expressivo e eficiência. Programas escritos em OCaml tendem a ter desempenho mais próximo de linguagens como C++, mas com código mais sucinto, geralmente mais próximo de linguagens como Python e Ruby.

Desempenho e expressividade

O gráfico na figura adiante mostra os dados do *Computer Language Benchmarks Game*, um *benchmark* comparativo que consiste em executar programas equivalentes em várias implementações de linguagens de programação. O gráfico mostra o tempo de execução dos programas no eixo horizontal, e o tamanho do código (em linhas) no eixo vertical (ambos os eixos estão em escala logarítmica para facilitar a visualização). Cada ponto representa um programa do *benchmark*, e os asteriscos grandes representam as médias de cada linguagem. Os dados podem ser encontrados em http://benchmarksgame.alioth.debian.org/.

Pelas médias, vemos que o desempenho dos programas em

OCaml é bastante próximo dos programas em C++, mas o tamanho dos programas também. Comparações baseadas nesse tipo de programa pequeno frequentemente apontam resultados que não ocorrem muito em programas reais. Nos dados da figura adiante, os programas em OCaml e C++ são, na média, cerca de 40 vezes mais rápidos que em Python e Ruby; essa diferença tende a ser menor na prática.

Da mesma forma, a experiência de programar em OCaml mostra que o tamanho do código deveria ser mais próximo do nível de Python e Ruby. Um fato que pode explicar a diferença é que a biblioteca padrão OCaml vem com menos funcionalidades predefinidas que as de Python e Ruby; isso pode aumentar o tamanho dos programas OCaml nessa comparação direta de programas pequenos, mas não faria muita diferença em programas maiores.

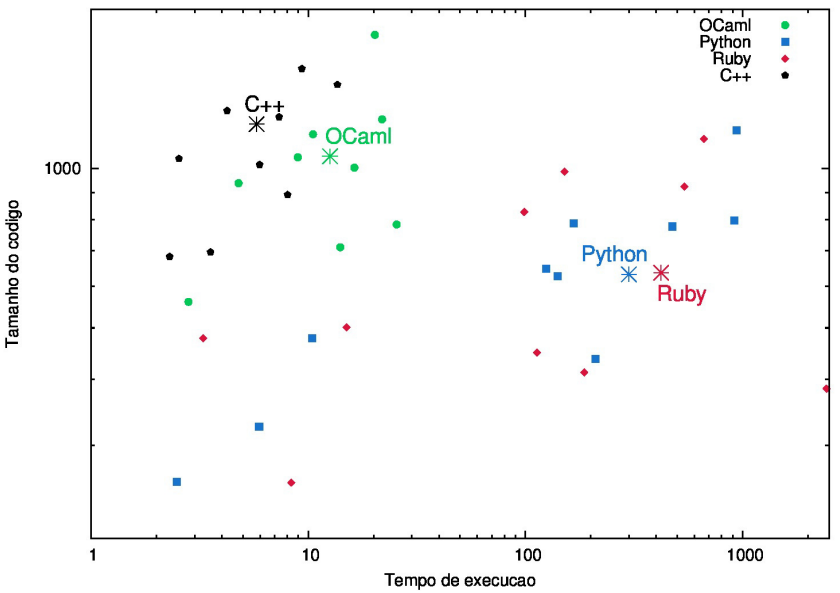

Figura 1.2: Desempenho e tamanho de código para dez programas do shootout em quatro linguagens

Uma experiência realista interessante foi a conversão do programa `0install` de Python para OCaml. O 0install é um sistema multiplataforma de instalação
para programas, e em junho de 2013 o seu autor, Thomas Leonard, decidiu estudar a viabilidade de várias linguagens para reescrever o programa, substituindo Python. Após análise, ele decidiu usar OCaml e portar para essa linguagem as mais de 30 mil linhas de código.

O programa resultante em OCaml tem aproximadamente o mesmo tamanho da versão original em Python (com uma pequena vantagem para o código OCaml), e é cerca de dez vezes mais rápido, na média. O autor gostou tanto da experiência de usar OCaml que hoje trabalha profissionalmente com a linguagem.

Os textos dele sobre a conversão do 0install para OCaml são interessantes para mostrar o ponto de vista de alguém interessado em usá-la. Uma retrospectiva que contém *links* para os outros textos que ele escreveu sobre o assunto pode ser encontrada em: http://roscidus.com/blog/blog/2014/06/06/python-to-ocaml-retrospective/.

Outras linguagens funcionais

É natural comparar a linguagem OCaml a outras linguagens funcionais. Algumas das mais conhecidas e usadas são Haskell, Scala, Clojure, F# e Scheme. Todas estas são interessantes para aprender e podem ser úteis em uma variedade de situações práticas.

Haskell é uma linguagem funcional pura e com avaliação não-estrita, o que exige o uso de métodos de programação bastante únicos que podem providenciar vários *insights* sobre a prática da programação. Clojure e Scheme são linguagens da família Lisp, que são interessantes pela maleabilidade do sistema de metaprogramação e da ideia de representar código e dados da

mesma forma. F# tem acesso às bibliotecas da plataforma .NET e do Mono, incluindo várias boas opções para desenvolvimento de jogos (MonoGame, Unity e o Unreal Engine, por exemplo).

Ainda assim, OCaml é uma boa opção para aprender como primeira linguagem funcional. Um dos motivos para isso é que OCaml é multiparadigma e permite o uso de características imperativas, ou mesmo orientadas a objetos, de maneira similar às linguagens mais conhecidas. Isso possibilita uma transição mais fácil para a programação funcional, ou mesmo a opção de usar código imperativo ou OO quando for a melhor solução.

Ao mesmo tempo, muitos dos conceitos de programação funcional da linguagem OCaml possuem similares próximos em outras linguagens funcionais, especialmente nas linguagens com tipagem estática como Haskell. Assim, um programador familiar com OCaml deve ter maior facilidade em aprender Haskell do que alguém que nunca trabalhou com uma linguagem funcional antes.

Por outro lado, começar com Haskell como primeira linguagem funcional tende a ser mais difícil do que começar com OCaml, pois a quantidade de novos conceitos a aprender é maior. Diferente da maioria das outras linguagens, Haskell tem avaliação *não-estrita* e usa o conceito de *mônadas* para controlar os efeitos colaterais.

Aprender F# para quem sabe OCaml é simples, já que as duas linguagens estão fortemente relacionadas. Os conceitos funcionais se aplicam de maneira similar em linguagens como Scala e Clojure, e mesmo em linguagens mais recentes como Idris.

OCaml e F#

As linguagens OCaml e F# são bastante similares, e não é à toa: F# foi basicamente criada para ser uma versão de OCaml para a plataforma .NET. Inicialmente criada como uma linguagem experimental por Don Syme, na Microsoft Research (braço da Microsoft ligado à pesquisa), F# rapidamente começou a achar usuários dentro da própria Microsoft, o que estimulou a continuação de seu desenvolvimento.

F# começou com um núcleo quase idêntico ao de OCaml, embora já com algumas diferenças; e com o tempo, foi adquirindo características que não existiam originalmente em OCaml, mesmo sem ter adquirido todos os aspectos da linguagem original. OCaml também continuou a ser desenvolvida e ganhou novas facetas que até hoje não existem F#.

A situação hoje é que ambas são quase idênticas nos seus núcleos básicos, mas cada uma possui uma série de características mais avançadas que a outra não tem. Além disso, existem diferenças importantes na questão das bibliotecas: F# tem acesso fácil às bibliotecas da plataforma .NET e do Mono, enquanto que OCaml tem acesso mais direto a bibliotecas criadas na linguagem C. Esses fatores fazem com que, na prática, código OCaml e F# tenham diferenças significativas em projetos reais, apesar de as duas linguagens terem núcleos tão similares.

OCaml possui um sistema de tempo de execução (*runtime*) relativamente simples, principalmente quando comparado a plataformas complexas como a JVM e o CLR da plataforma .NET e

do Mono. O compilador OCaml funciona de maneira relativamente transparente quanto ao desempenho do código gerado: a relação entre o código e seu desempenho quando compilado é relativamente previsível. Ou seja, um programador OCaml com alguma experiência consegue ter uma boa ideia se um trecho de código vai executar rapidamente ou não.

Esses fatos são importantes em aplicações que fazem interface com sistemas mais básicos, como sistemas embarcados. Isso também significa que saber como o *runtime* representa os valores de OCaml na memória torna o interfaceamento com bibliotecas criadas na linguagem C uma tarefa simples.

1.4 USOS E APLICAÇÕES

OCaml certamente é menos usada do que linguagens conhecidas, como C++ e Java, mas está longe de ser apenas uma linguagem experimental. Existem usuários com aplicações de complexidade e tamanho significativos escritas na linguagem. Muitas dessas aplicações tendem a ser de domínios relativamente complexos ou que usam pesquisa de ponta, e algumas delas funcionam em um ambiente que requer alta confiabilidade. Nesta seção, mencionamos alguns dos usos, usuários e aplicações importantes de OCaml.

A **Jane Street Capital** é uma empresa financeira especializada na compra e venda de produtos financeiros em alta frequência (*High-Speed Trading*). A Jane Street usa OCaml como sua linguagem principal em toda a empresa, inclusive nos programas que realizam altos volumes de transações financeiras.

Como essas transações ocorrem muito rapidamente, defeitos no código dos programas podem causar prejuízos enormes em pouquíssimo tempo; isso exige software de alta confiabilidade. Após

um tempo usando linguagens mais conhecidas, a Jane Street escolheu OCaml para criar esses programas, considerando que essa é uma boa escolha devido aos requerimentos impostos pela atuação da empresa.

Atualmente, a Jane Street é um dos maiores usuários da linguagem, com uma base de código OCaml de mais de três milhões de linhas. Yaron Minsky da Jane Street explica a experiência deles usando OCaml no vídeo *Caml Trading* http://vimeo.com/14317442

Coq é um sistema assistente para provas (*proof assistant*) que pode ser usado para auxiliar na prova de teoremas matemáticos (e verificar as provas criadas), para verificação de outros sistemas de software (provar que um programa segue a sua especificação), e como sistema de programação, para criar os chamados *programas certificados*. Um programa certificado é um programa correto por construção, ou seja, que já é criado com uma prova de corretude associada. Simplificando, a ideia é que, se um programa escrito em Coq usando *especificações fortes* é compilado corretamente (embora não seja exatamente um compilador), temos uma garantia de que o programa segue a sua especificação.

Coq já foi usado na verificação de importantes teoremas matemáticos (como o teorema das quatro cores), na escrita de um compilador certificado para a linguagem C (http://compcert.inria.fr/) e na verificação de vários sistemas de missão crítica. O sistema Coq em si é o resultado de três décadas de pesquisa no INRIA e é escrito em OCaml, incluindo cerca de 200 mil linhas de código na linguagem. Em 2013, o sistema Coq ganhou o prêmio *Software System Award* da ACM (*Association for Computing Machinery*) como reconhecimento por ser um sistema de software com influência duradoura.

O **Mirage OS** é um sistema operacional criado especificamente para a computação em nuvem, e é escrito completamente em

OCaml. O site do sistema operacional é http://www.openmirage.org/, e funciona a partir de um servidor em que todo o software foi escrito em OCaml, incluindo os protocolos de rede.

Para criar uma aplicação que rodará na nuvem, o usuário cria um programa OCaml que usa as bibliotecas do Mirage. Durante o desenvolvimento, essas APIs funcionam como bibliotecas que executam em sistemas Unix, e o programa compilado pode ser testado como qualquer outro. Quando é necessário fazer *deploy* do serviço ou aplicação, o programa em OCaml é compilado para um *unikernel*, um núcleo de sistema operacional criado especificamente para a aplicação; as chamadas à API do Mirage nesse contexto viram chamadas de sistema. A aplicação criada dessa forma pode executar em qualquer plataforma que suporte o hipervisor Xen, o que inclui a maior parte dos fornecedores de servidores na nuvem, como Amazon EC2, Linode, Rackspace e outros.

Um dos problemas que a Microsoft tinha com o Windows era o grande número de erros críticos que o sistema apresentava (a famosa tela azul, por exemplo). Analisando os dados relativos a esses erros, ficou claro que a maior parte dos problemas era causada por falhas nos drivers de dispositivos, muitos deles criados por outras empresas e não pela própria Microsoft.

Essa descoberta estimulou a criação de um projeto para detecção automática de defeitos e verificação de drivers escritos em C. Esse projeto se chamou SLAM e um dos produtos resultantes dele foi a ferramenta SDV (*Static Driver Verifier*), que foi criada para a verificação de drivers. O SDV foi escrito em OCaml e foi incluído no Kit de desenvolvimento para drivers oficial da Microsoft (*Windows Driver Development Kit*), e essa tecnologia foi importante na diminuição dos erros críticos em versões mais recentes do Windows.

Além do compilador da linguagem Hack, o **Facebook** tem várias ferramentas internas criadas na linguagem OCaml. Um exemplo é o `pfff`, um conjunto de ferramentas para análise de código escrito em várias linguagens (incluindo OCaml e PHP); essas análises auxiliam na implementação, detecção automática de defeitos, indexação e busca de código, e visualização de código. As ferramentas `pfff` são usadas no código do próprio Facebook, uma base de mais de 5 milhões de linhas de código em PHP (e, hoje em dia, uma parte em Hack).

Esses são apenas alguns exemplos. Para encontrar outros, basta procurar pelas palestras apresentadas em eventos como o *OCaml Users and Developers Workshop* (OUD) e o *Commercial Users of Functional Programming* (CUFP).

1.5 O SISTEMA OCAML

O sistema OCaml é um conjunto de ferramentas criadas para programar em OCaml, incluindo os compiladores e o REPL. Além do sistema básico, criado e mantido pela INRIA, a maioria dos programadores OCaml utiliza um conjunto de ferramentas e bibliotecas externas que tornam o uso da linguagem mais prático. Uma delas é o gerenciador de pacotes OPAM (*OCaml PAckage Manager*), que serve tanto para instalar o próprio sistema OCaml quanto bibliotecas criadas por terceiros.

Nesta seção, o objetivo é entender, em linhas gerais, como as ferramentas para programação OCaml estão organizadas, quais não são parte do sistema básico são recomendadas para uso, e como instalá-las.

Ferramentas da distribuição

A distribuição OCaml básica inclui um conjunto de ferramentas

importantes para a programação na linguagem. Além do REPL (`ocaml`), do compilador nativo (`ocamlopt`) e do compilador de *bytecode* (`ocamlc`), algumas outras ferramentas importantes são:

- `ocamldoc` é um gerador de documentação baseado em comentários, similar ao JavaDoc.
- `ocamlbuild` é uma ferramenta para compilação de projetos OCaml.
- `ocamldep` é um analisador de dependências entre arquivos de código-fonte, geralmente usado para determinar a ordem de compilação dos arquivos.
- `ocamllex` e `ocamlyacc` são geradores de analisadores léxicos e sintáticos, usados principalmente na criação de compiladores e interpretadores para outras linguagens.
- `ocamlmktop` possibilita a criação de REPLs custommizados para OCaml. As customizações incluem, por exemplo, carregar certas bibliotecas automaticamente para uso no modo interativo.

A distribuição também inclui algumas outras ferramentas, mas as duas primeiras (`ocamldoc` e `ocamlbuild`) são as mais utilizadas.

REPL e uso interativo

Usar a linguagem em modo interativo pode ser bem útil, principalmente para testar e explorar novas ideias. A possibilidade de obter respostas imediatamente, sem precisar compilar um arquivo e executá-lo na linha de comando, é uma conveniência que muitos usuários consideram imprescindível.

Esse uso interativo é especialmente útil para testar os trechos de código que são apresentados como exemplo neste livro. Poder

reproduzir os exemplos, tentar variações e ver imediatamente os resultados tendem a acelerar o aprendizado. Para usar a linguagem interativamente, a forma mais comum é usar algum tipo de REPL (*Read Eval Print Loop*).

O REPL padrão incluído com a distribuição da linguagem (`ocaml`) funciona, mas tem muitas deficiências importantes. A edição da entrada é bastante básica e não permite, por exemplo, usar as teclas direcionais para mudar o local do cursor. Também não há suporte para um histórico de linhas digitadas. Uma solução é o utilitário `ledit` para adicionar essas capacidades ao REPL padrão.

Melhor ainda é usar o `utop` , um REPL bastante melhorado para a linguagem, que inclui o uso de cores no terminal e suporte a autocompletar. Instalá-lo é fácil para quem usa o OPAM, como veremos a seguir. Outra opção é o `ocaml-top` , que usa uma interface gráfica com o usuário.

Pacotes e bibliotecas: OPAM e findlib

Duas ferramentas externas ajudam na instalação e uso de bibliotecas e outros pacotes para a linguagem OCaml: `OPAM` (*OCaml Package Manager*) e `findlib` . `findlib` é uma ferramenta para instalação, gerenciamento e uso de bibliotecas para a linguagem OCaml. Os compiladores OCaml acessam bibliotecas de forma similar aos compiladores das linguagens C e C++, usando opções de linha de comando para incluir certos diretórios onde o compilador deve buscar os arquivos necessários. O `findlib` inclui uma ferramenta chamada `ocamlfind` que simplifica bastante o uso das bibliotecas nos compiladores e outras ferramentas do sistema OCaml.

`findlib` funciona apenas localmente, como um registro central de bibliotecas e uma forma de usá-las facilmente. O cenário de uso apenas do `findlib` é obter o código-fonte de uma

biblioteca, compilar o código, e instalar a biblioteca compilada, isso tudo feito pelo usuário. Ele não baixa pacotes da internet, não gerencia as dependências, não compila os pacotes, nem sabe como atualizar as bibliotecas. Para isso, é necessário usar o OPAM.

`OPAM` é um sistema de gerenciamento de pacotes para a linguagem OCaml com um sistema de gerenciamento de dependências avançado. Apesar de ser uma ferramenta recente em comparação a outras do ecossistema OCaml, hoje a maioria dos usuários da linguagem usa o `OPAM`. Com ele, é possível instalar bibliotecas (e suas dependências) automaticamente, e mantê-las atualizadas. Também é possível ter várias versões do sistema OCaml instaladas, e selecionar qual versão se quer usar (isso é usado inclusive para testar versões futuras do compilador e novas *features*).

O `OPAM` se encarrega de baixar o código-fonte, compilar e instalar a biblioteca; esse processo de instalação inclui instalar a biblioteca no registro de bibliotecas do `findlib`. O uso das bibliotecas continua sendo através do `findlib`. Também é fácil publicar seus próprios pacotes usando o `OPAM`.

Instalação

O sistema OCaml está disponível para Windows, Mac OS X, Linux e FreeBSD, e pode funcionar em outros sistemas Unix. Instruções que indicam a melhor forma de instalar em cada sistema podem ser encontradas em http://ocaml.org/docs/install.html.

Basicamente, em sistemas Linux, o mais adequado é instalar o sistema OCaml usando o gerenciador de pacotes da distribuição; a maioria das distribuições mais usadas tem um pacote com o sistema OCaml. Em algumas distribuições (por exemplo, versões recentes de Debian e Ubuntu), também é possível instalar o `OPAM` pelo gerenciador de pacotes. Caso ele não esteja disponível, as instruções

para instalar estão em http://opam.ocaml.org/doc/Install.html.

No Mac OS X, é recomendável usar o *homebrew*, que contém pacotes para o sistema OCaml e para o `OPAM`, e eles costumam ser bem mantidos e atualizados. O comando a seguir é suficiente para instalar o sistema OCaml e o `OPAM`.

```
brew install opam
```

Independente do sistema operacional, com as ferramentas OCaml e o `OPAM` instalados, para instalar o `utop` basta usar, na linha de comando:

```
opam install utop
```

Para quem prefere uma interface gráfica, existe o `ocaml-top`:

```
opam install utop
```

Infelizmente, o OPAM ainda não suporta o uso em sistemas Windows, mas o `findlib` e muitas bibliotecas importantes funcionam bem nele. Uma forma de instalar o sistema OCaml é usando o Cygwin, que fornece vários componentes de um sistema Unix no Windows. Dependendo de quais pacotes sejam instalados no Cygwin, os programas OCaml compilados poderão precisar da DLL do Cygwin para executar.

Um instalador que automatiza a instalação do sistema OCaml e vários outros componentes da melhor maneira possível no Windows (incluindo a geração de executáveis que não dependem da DLL do Cygwin) pode ser encontrado em https://protz.github.io/ocaml-installer/.

Um instalador criado pela OCamlPro e que inclui algumas extensões proprietárias para funcionar melhor no Windows é o OCPWin (http://typerex.ocamlpro.com/ocpwin.html).

Instalar o utop no Windows é possível, mas é preciso compilar

várias bibliotecas das quais o utop depende. Uma alternativa mais interessante é instalar o `ocaml-top` (http://typerex.ocamlpro.com/ocaml-top.html).

Editores

Existe suporte de boa qualidade para programação na linguagem OCaml em vários editores de código. Os tradicionais `Emacs` e `vim` incluem modos para a linguagem OCaml; na verdade, existem dois modos para usar OCaml no Emacs, `caml-mode` e `tuareg`, com muitos usuários preferindo a segunda opção.

Uma outra opção popular para edição de texto é Sublime Text. Para quem usa o *Package Control* no Sublime, é recomendado buscar e instalar o pacote de nome *OCaml* em vez de usar o suporte *default* para a linguagem. Também existem pacotes de suporte para outros editores como Textmate e Atom, assim como a maioria dos outros editores usados por programadores.

Material oficial

O site mais indicado como página principal para OCaml é o http://ocaml.org.

O INRIA, instituição que criou e atua como principal mantenedor da linguagem, tem um site com menos informações, em http://caml.inria.fr.

A versão estável mais recente do manual de referência oficial de OCaml pode ser encontrada em http://caml.inria.fr/pub/docs/manual-ocaml/.

1.6 ORGANIZAÇÃO DO LIVRO

Podemos organizar os capítulos do livro em basicamente duas

partes: a primeira (capítulos 2 a 7) é uma introdução à linguagem e descrição dos seus aspectos fundamentais, incluindo as suas características funcionais e imperativas. No meio dessa parte, há um capítulo com um exemplo mais substancial: o capítulo 6 mostra a criação de um interpretador, um compilador e uma máquina virtual simples.

A segunda parte introduz um conjunto de características mais avançadas de OCaml, começando pelo sistema de módulos no capítulo 8, e continuando com rótulos, variantes polimórficas e objetos. No meio desta parte, também há um capítulo com um exemplo maior: no capítulo 9, vemos como analisar dados com OCaml e como criar, automaticamente, classificadores de dados empregando a técnica das árvores de decisão.

O objetivo de cobrir esses tópicos é possibilitar ao leitor escrever programas OCaml, e ler e entender a maior parte (de 80% a 90%) do código OCaml disponível na internet. A partir daí, o leitor pode continuar se aprofundando na linguagem com outros materiais, consultas ao manual de referência etc. Materiais adicionais serão publicados também no site do livro em http://andreiformiga.com/livro/ocaml.

Capítulo 2

TIPOS E VALORES BÁSICOS

Vamos começar a explorar a linguagem OCaml usando uma sequência de pequenos exemplos que demonstram suas características mais fundamentais, incluindo como definir variáveis e funções, os seus tipos simples e as operações mais comuns com valores destes tipos.

2.1 PRIMEIROS PASSOS

Neste capítulo, vamos usar o REPL para verificar rapidamente o resultado de algumas expressões básicas de OCaml. O REPL padrão tem alguns problemas de usabilidade que podem ser resolvidos com a instalação da ferramenta `ledit`.

Uma alternativa recomendável é usar o `utop`, um REPL melhorado para OCaml. Para instalar o `ledit` ou `utop`, verifique as instruções de instalação na seção *O sistema OCaml* (capítulo anterior). Outra possibilidade é usar um modo interativo para a linguagem OCaml na Web, o TryOCaml, em http://try.ocamlpro.com/. Entretanto, principalmente para os capítulos seguintes do livro, é recomendável instalar o sistema OCaml localmente.

Ao iniciar o REPL, aparece como *prompt* um caractere `#`. Para não fugir da tradição, vamos começar com um *Hello, world!*. No trecho a seguir, a primeira linha é digitada pelo usuário no REPL, a segunda é o *Hello, world!* impresso, e a terceira é o resultado da

expressão (isso será explicado mais tarde).

```
# print_string "Hello, world!\n";;
Hello, world!
- : unit = ()
```

Note que a entrada do usuário só é processada pelo REPL (padrão ou `utop`) após a digitação de dois caracteres ponto e vírgula (`;;`). Esses caracteres não são necessários para o código-fonte escrito em arquivos (embora possam ser usados em arquivos), mas sim como um separador para o REPL. Os exemplos deste livro cuja linha começa com `#` estão mostrando um trecho de uma sessão de uso do REPL; o usuário não deve digitar o `#`, mas o que vem depois dele, incluindo os dois caracteres ponto e vírgula.

Nesse caso, também é mostrada a saída do REPL, embora parte seja omitida quando ela é muito longa. Exemplos que não começam `#` podem ser pensados como trechos de código que estariam em algum arquivo, e geralmente não incluem os dois pontos e vírgulas nem mostram nenhuma saída.

O REPL pode ser usado como uma calculadora (mais uma vez, a primeira linha deve ser entrada pelo usuário, e a segunda é a resposta do REPL):

```
# 3 + 7 * 2;;
- : int = 17
```

Comentários em OCaml são iniciados com `(*` e terminados com `*)`:

```
# (* computador, por favor some 2 mais 3 *) 2 + 3;;
- : int = 5
```

A resposta do REPL mostra o valor da expressão, 5, mas também o seu tipo, `int`. Essa é uma demonstração simples da *inferência de tipos* da linguagem.

2.2 VARIÁVEIS E TIPOS BÁSICOS

Variáveis podem ser declaradas usando a palavra-chave `let`:

```
# let x = 7;;
val x : int = 7

# let resposta = x * 6;;
val resposta : int = 42
```

Também podemos definir variáveis de outros tipos:

```
# let s = "Aussonderungsaxiom";;
val s : bytes = "Aussonderungsaxiom"

# let pi = 3.14159265359;;
val pi : float = 3.14159265359

# let b1 = 1 < 2;;
val b1 : bool = true
```

OS TIPOS STRING E BYTES

Até recentemente só havia um tipo para strings em OCaml, o tipo `string`. Valores do tipo `string` são mutáveis, ou seja, é possível alterar caracteres individuais de cada uma, assim como acontece em linguagens como C e C++, mas diferente de outras, como Java e C#.

A mutabilidade das strings em OCaml existiu principalmente para possibilitar seu uso como *buffers* em situações em que uma sequência de *bytes* era necessária. Por exemplo, em algumas funções de entrada e saída de baixo nível (como no módulo `Unix` da biblioteca padrão), é preciso passar um *buffer* que será preenchido pela função, como resultado da leitura de informações de um arquivo em disco ou de outra fonte. *Buffers* desse tipo também são frequentemente necessários para chamar funções de bibliotecas criadas na

linguagem C.

Mas strings mutáveis impõem uma série de desvantagens para o *runtime* da linguagem e eliminam algumas possibilidades de otimização pelo compilador. Além disso, ter um tipo string mutável por padrão, sem uma versão imutável do mesmo tipo, faz com que não haja uma opção mais confiável para programas OCaml que precisem de mais garantias de corretude.

Por isso, as versões recentes da linguagem têm dois tipos para strings: o tipo `string` e o tipo `bytes`. Por enquanto, os dois são sinônimos por padrão, mas existe uma opção de compilação que faz com que o tipo `string` tenha valores imutáveis, e o tipo `bytes` seja igual ao tipo `string` antigo. A ideia é que isso seja usado durante um período de transição, e futuramente esse será o comportamento padrão do compilador: `string` é para strings imutáveis e `bytes` para strings mutáveis. Atualmente, o REPL tende a identificar o tipo de literais string como `bytes`.

Nomes de variáveis em OCaml devem começar com uma letra minúscula ou *underscore*, potencialmente seguida por letras (maiúsculas ou minúsculas), dígitos, *underscores* e aspas simples. O valor de uma variável já definida pode ser obtido no REPL apenas digitando seu nome:

```
# resposta;;
- : int = 42

# let pi = 3.14159265359;;
val pi : float = 3.14159265359
```

Se pedirmos ao REPL pelo nome de uma variável não definida, aparece um erro:

```
# pergunta;;
Error: Unbound value pergunta
```

Já vimos alguns dos principais tipos básicos da linguagem: `int`, `float`, `string` (ou `bytes`) e `bool`. O tipo `char`, como esperado, é usado para representar caracteres:

```
# let c = 'A';;
val c : char = A
```

Caracteres em OCaml ocupam um *byte* de memória e seguem o código ASCII. Suporte ao uso de Unicode existe em bibliotecas externas (*Camomile* é uma das mais usadas).

Operações com inteiros

As operações aritméticas usuais estão disponíveis para os inteiros: soma, subtração, multiplicação e divisão. A divisão neste caso é inteira e retorna apenas o quociente:

```
# 11 / 3;;
- : int = 3
```

O resto de uma divisão inteira pode ser obtido com o operador de resto, `mod`:

```
# 11 mod 3;;
- : int = 2
```

A precedência e associatividade dos operadores inteiros seguem as convenções usuais da matemática e outras linguagens de programação. Para detalhes, consultar a Seção 6.7 do manual da linguagem, que pode ser acessada no endereço http://caml.inria.fr/pub/docs/manual-ocaml/expr.html.

Operações com números de ponto flutuante

As operações usuais com valores `float` também estão disponíveis, mas OCaml é mais rígida com os tipos numéricos do

que a maioria das linguagens: em OCaml nunca ocorrem conversões ou promoções automáticas de valores numéricos para outros tipos. Tentar multiplicar um `int` por um `float` vai resultar sempre em um erro de tipo; é necessário converter explicitamente um dos dois valores para o tipo do outro.

Além disso, os próprios operadores em OCaml são diferenciados para quando se aplicam em inteiros ou `float` s. Já vimos os operadores inteiros das quatro operações aritméticas básicas; a versão `float` destes é seguida por um ponto:

```
# 3.0 *. 4.5 +. 2.2;;
- : float = 15.7

# (11.0 /. 3.0) -. 2.0;;
- : float = 1.66666666666666652
```

Note o uso dos operadores `+.` , `*.` , `-.` e `/.` no exemplo anterior. A separação dos operadores que funcionam com inteiros e números de ponto flutuante é uma frequente fonte de reclamações para iniciantes na linguagem, mas na prática incomoda menos do que parece.

O uso de algumas técnicas ligadas aos módulos (ver capítulo *Módulos*) pode possibilitar o uso dos operadores mais tradicionais para operações com números de ponto flutuante. Em versões futuras da linguagem OCaml, isso poderá ser resolvido ainda mais facilmente com o uso de *implícitos modulares*, uma característica que está atualmente em fase experimental para inclusão na linguagem.

Para converter um número inteiro para `float` , pode-se usar a operação também chamada `float` . No exemplo a seguir, a primeira tentativa resulta em um erro de tipo, pois tenta dividir um inteiro por um `float` . Em seguida, a conversão explícita para `float` resolve o problema:

```
# 11 /. 3.0;;
Error: This expression has type int but an expression
was expected of type float

# (float 11) /. 3.0;;
- : float = 3.66666666666666652
```

A conversão de `float` para `int` pode ser feita truncando a parte após o ponto com a operação `truncate`:

```
# truncate 3.14159265358979312;;
- : int = 3
```

Um operador predefinido para `float` que não tem similar nos inteiros em OCaml é a exponenciação, representada com dois asteriscos:

```
 # 2.0 ** 8.0;;
- : float = 256.
```

Mais uma vez, a precedência e associatividade dos operadores `float` não devem apresentar surpresas, mas os detalhes podem ser consultados no manual.

Operações booleanas

Para lidar com valores booleanos, existem as operações usuais: `E` (conjunção), `OU` (disjunção) e `NÃO` (negação). Os operadores são `&&`, `||` e `not`, respectivamente:

```
# true && false;;
- : bool = false

# false || not true;;
- : bool = false
```

Comparações entre valores têm resultado booleano. Com relação às comparações, a forma mais indicada de comparar por igualdade em OCaml é usando o operador `=` para igualdade, e `<>` para desigualdade:

```
# 2 = 2;;
```

```
- : bool = true

# 2 = 3 || 2 <> 3;;
- : bool = true
```

Os outros operadores de comparação são os usuais: `>` , `<` , `<=` e `>=` . A avaliação de expressões booleanas em OCaml segue a estratégia conhecida como *curto-circuito*: a avaliação termina assim que o valor da expressão completa pode ser determinado, mesmo que a expressão completa não seja avaliada.

Por exemplo, na expressão `true || x = y` , o compilador não precisa verificar o valor das variáveis `x` e `y` , pois o resultado da expressão já é conhecido como `true` . Isso é mais importante na presença de efeitos colaterais, como discutido no capítulo *Características imperativas*.

A expressão condicional (`if` / `then` / `else`) usa um valor booleano para decidir seu resultado. Diferente das linguagens imperativas em que existe um *comando* condicional, a expressão condicional pode ser usada em qualquer contexto onde uma expressão pode ocorrer:

```
# let x = if 3 > 4 then 3 else 4;;
val x : int = 4
```

Isso é uma consequência de a linguagem OCaml ser *orientada a expressões*, como discutido no capítulo *Introdução*. Uma consequência de o `if` ser uma expressão é que ambos os possíveis resultados (após `then` e `else`) devem ter o mesmo tipo. Caso contrário, uma atribuição como a do exemplo a seguir não poderia determinar o tipo da variável `y` :

```
# let y = if 3 > 4 then 3 else 4.0;;
Error: This expression has type float but an expression
was expected of type int
```

Apesar de parecer limitador, isso raramente cria problemas na prática. É possível ter um `if` sem a parte `else` , mas essa é uma

característica imperativa e será vista no capítulo *Características imperativas*.

Operações básicas com strings

O operador `^` concatena duas strings:

```
# let inigo = "Meu nome eh " ^ "Inigo Montoya";;
val inigo : bytes = "Meu nome eh Inigo Montoya"
```

Caracteres individuais de uma string `s` podem ser acessados usando a sintaxe `s.[i]`, onde `i` é o índice. Note o uso de um ponto entre o nome e o colchete. O primeiro caractere tem índice zero e, se o índice estiver fora da string, ocorre uma exceção:

```
# inigo.[4];;
- : char = 'n'

# inigo.[35];;
Exception: Invalid_argument "index out of bounds".
```

2.3 FUNÇÕES

Para definir uma função, também se usa `let`, seguido pelo nome da função (seguindo a mesma convenção léxica para nomes de variáveis) e os nomes de um ou mais parâmetros separados por espaço:

```
# let quadrado x = x * x;;
val quadrado : int -> int = <fun>
```

Mais uma vez, o REPL responde com o valor recém-definido e seu tipo: a segunda linha diz que `quadrado` tem tipo `int -> int` e é uma função (indicada como `<fun>`). Em OCaml, não há diferença entre uma variável inteira e uma função, como ficará mais claro no capítulo *Programação funcional*; a única diferença para o resultado mostrado na definição de uma variável é que o valor da função não é mostrado, apenas indicado com `<fun>`. Isso ocorre

pois o código da função é compilado imediatamente (como vimos na seção *Características de OCaml*), e o código-fonte original da função não é armazenado.

Note também que o REPL inferiu automaticamente o tipo da função: o tipo `int -> int` representa as funções que recebem um parâmetro do tipo `int` e retornam um `int`. A inferência funciona de acordo com o uso do parâmetro da função: como `x` é multiplicado por ele mesmo, e a multiplicação `*` é um operador que funciona com inteiros (já vimos que o operador de multiplicação para `float`s é diferente: `*.`), `x` deve ser do tipo `int`, e a multiplicação de `int`s resulta em um `int`, que é o tipo de retorno da função.

Para chamar a função passando um argumento, é só usar o nome da função seguido do argumento, separado por pelo menos um espaço:

```
# quadrado 5;;
- : int = 25
```

A sintaxe de chamada de funções em OCaml é diferente da maioria das linguagens mais conhecidas, pois não é necessário usar parênteses para especificar o argumento.

```
# quadrado(5);;
- : int = 25
```

Embora nesse caso esse código funcione, isso é apenas um efeito do fato de a função só ter um parâmetro, e não vai funcionar com funções com mais parâmetros. Na verdade, `quadrado(5)` é interpretado em OCaml como `quadrado (5)`, aplicação da função `quadrado` a um argumento que é a expressão `(5)`. Como adicionar parênteses a uma expressão não muda seu valor, isso é o mesmo que `quadrado 5`.

Para definir uma função de dois ou mais parâmetros, é só incluí-

los após o primeiro, separados por espaços:

```
# let mult x y = x * y;;
val mult : int -> int -> int = <fun>
```

A chamada a uma função com vários parâmetros é similar:

```
# mult 4 5;;
- : int = 20
```

Neste caso, tentar usar a sintaxe de chamada de função de outras linguagens não funciona:

```
# mult(4, 5);;
Error: This expression has type 'a * 'b but
an expression was expected of type int
```

O tipo de `mult` é `int -> int -> int`, o que significa que é uma função com dois parâmetros do tipo inteiro e que retorna um inteiro. O tipo após a última seta é o de retorno, e os anteriores são os dos parâmetros. Os tipos de funções serão explicados em mais detalhes no capítulo *Programação funcional*.

Definindo operadores

Em muitas linguagens de programação, existe a possibilidade de fazer *sobrecarga de operadores*, ou seja, alterar o significado de operadores como `+` e `*`. Em OCaml, operadores são apenas funções, mas que são chamadas com notação *infixa*.

Normalmente, são chamadas em notação *pré-fixada*, ou seja, a função vem antes dos argumentos: `f x y`, onde `f` é a função, e `x` e `y` são argumentos. É possível definir funções em OCaml para serem chamadas em notação infixa, ou seja, entre os argumentos: `x f y`.

Podemos ver que operadores como `+` são realmente funções em OCaml: para se referir a um operador como um identificador da linguagem, é só colocá-lo entre parênteses. Se fizermos isso no

REPL, o tipo do operador é impresso, assim como ocorre para outras variáveis e funções:

```
# (+);;
- : int -> int -> int = <fun>

# (+.);;
- : float -> float -> float = <fun>
```

Da mesma forma, podemos declarar novos operadores usando parênteses na declaração:

```
# let (++) x y = x + y + y;;
val ( ++ ) : int -> int -> int = <fun>

# 5 ++ 7;;
- : int = 19
```

Os caracteres que podem ser usados para criar operadores são:

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

Operadores que começam com `!`, `?` e `~` são considerados *pré-fixos*; os demais são considerados *infixos*, como o `++` do exemplo anterior. É recomendável não defini-los começando com dois pontos `:`, pois o analisador sintático trata esse caractere de maneira especial (é um operador de listas, como veremos no capítulo *Registros e variantes*).

A precedência e associatividade dos operadores definidos depende do primeiro caractere usado no operador, na maioria dos casos. Por exemplo, os que começam com `*` sempre têm precedência mais alta do que aqueles que começam com `+`. A especificação completa pode ser vista em uma tabela na Seção 6.7 (*Expressions*) no manual oficial da linguagem: http://caml.inria.fr/pub/docs/manual-ocaml/expr.html.

É possível inclusive redefinir os operadores predefinidos da linguagem OCaml. Um detalhe com relação a operadores que usem o caractere `*` é lembrar de que `(*` inicia um comentário em

OCaml, e `*)` fecha um comentário. Logo, na hora de declarar operadores, deve-se evitar usar um asterisco `*` junto de um parêntese. P

ara evitar problemas, basta adicionar um ou mais espaços separando o asterisco dos parênteses. Por exemplo, para redefinir o operador `*` para multiplicar `float`s em vez de inteiros:

```
# let ( * ) x y = x *. y;;
val ( * ) : float -> float -> float = <fun>
```

Alguns operadores predefinidos são considerados como palavras-chaves da linguagem e podem não se comportar bem quando predefinidos. A lista completa de operadores considerados como palavras-chaves pode ser encontrada no manual da linguagem, na Seção 6.1, *Lexical conventions*, subseção *Keywords*: http://caml.inria.fr/pub/docs/manual-ocaml/lex.html

Alterar o significado de um operador predefinido globalmente em um programa não é, geralmente, uma boa ideia. Mas como os programas em OCaml são organizados em termos de módulos, é comum definir novos operadores ou redefinir os predefinidos apenas no contexto de algum módulo específico, o que é bastante útil. Veremos no capítulo *Módulos* como usar operadores definidos em um módulo sem poluir o espaço de nomes global.

Declarações locais

Quando definimos uma variável ou função usando `let`, o resultado é uma definição que fica disponível em todo o módulo onde a declaração se encontra (sempre existe um módulo atual, mesmo que implícito). Às vezes, queremos definir uma variável ou função para ser usada apenas localmente dentro de uma expressão. Para isso, existe a forma `let ... in e`, que faz com que os nomes definidos no `let` estejam disponíveis apenas na expressão `e`.

No exemplo a seguir, a variável `x` só existe durante a expressão após o `in` da primeira linha, e não cria uma variável `x` no módulo inteiro:

```
# let x = 2 in x * x * 4;;
- : int = 16

# x;;
Error: Unbound value x
```

Se houvesse uma variável de nome `x` já existente no módulo, a variável local criada no `let/in` esconde a variável externa, sem alterar seu valor fora da expressão do `let/in`:

```
# let x = 1001;;
val x : int = 1001

# let x = 2 in x * x * 4;;
- : int = 16

# x;;
- : int = 1001
```

Na prática, isso quase sempre é usado para criar variáveis ou funções locais dentro de outras funções. Vamos dizer que precisamos de uma função que calcula a raiz positiva de uma equação de segundo grau, se houver (se não houver, retorna o valor *infinito*):

```
# let raiz_positiva a b c =
    if (b *. b -. 4.0 *. a *. c) >= 0.0 then
      (-.b +. (sqrt (b *. b -. 4.0 *. a *. c))) /. (2.0 *. a)
    else infinity;;
val raiz_positiva : float -> float -> float -> float = <fun>
```

Além da repetição do cálculo do discriminante na fórmula da equação de segundo grau, o cálculo da raiz com uma expressão muito longa torna o código mais difícil de entender. Usando uma variável local para o discriminante (normalmente chamado de *delta*), a mesma função fica mais legível:

```
# let raiz_positiva a b c =
```

```
    let delta = b *. b -. 4.0 *. a *. c in
    if delta >= 0.0 then
      (-.b +. sqrt delta) /. 2.0 *. a
    else infinity;;
val raiz_positiva : float -> float -> float -> float = <fun>
```

A formatação no exemplo anterior é típica em OCaml: a definição do `delta` em uma linha usando `let/in`, e o resto do código continuando na linha seguinte, com a mesma indentação. Para definições locais mais longas (por exemplo, funções locais), às vezes se usa uma convenção diferente. Veremos exemplos disso nos capítulos seguintes.

Funções recursivas

Começamos este capítulo mostrando um exemplo de *Hello, World!* porque esse é tradicionalmente o primeiro programa mostrado para linguagens imperativas. Nas linguagens funcionais, o *Hello, World!* é a função `fatorial`.

O fatorial de um número inteiro `n` é definido da seguinte forma: o fatorial de zero é igual a um, e o fatorial de um número `n` maior que zero é igual ao produto de todos os números de 1 até `n`. Se tentarmos escrever a função fatorial dessa forma, encontramos um problema:

```
# let fatorial n =
    if n = 0 then 1 else n * fatorial (n - 1);;
Error: Unbound value fatorial
```

O erro diz que a função `fatorial` usada no lado `else` do condicional não está definida. Mas isso ocorre porque estamos tentando definir a própria função `fatorial` em função dela mesma. Nesse caso, é preciso comunicar ao compilador OCaml que a função fatorial é recursiva, e portanto ela pode chamar ela mesma. Isso é feito usando a forma `let rec`:

```
# let rec fatorial n =
    if n = 0 then 1 else n * fatorial (n - 1);;
```

```
val fatorial : int -> int = <fun>
```

Agora podemos usar a função `fatorial`:

```
# fatorial 3;;
- : int = 6

# fatorial 5;;
- : int = 120
```

No caso de funções que apresentem recursividade mútua, todas elas devem ser definidas conjuntamente usando a forma `let rec ... and ...`. Um exemplo é escrever duas funções recursivas para calcular se um número inteiro positivo é par ou ímpar.

A estratégia é simples (embora pouco prática): consideramos zero e um como casos básicos, sendo zero par e um ímpar. Para outro número `n` (diferente de zero e um), sabemos que `n` é par se `n-1` é ímpar, e sabemos que `n` é ímpar se `n-1` é par. Dessa forma, as duas funções são mutuamente recursivas e é preciso defini-las da seguinte forma:

```
# let rec par n =
    if n = 0 then true
    else if n = 1 then false
    else impar (n-1)
  and impar n =
    if n = 0 then false
    else if n = 1 then true
    else par (n-1);;
val par : int -> bool = <fun>
val impar : int -> bool = <fun>
```

Veremos muitos outros exemplos de funções recursivas ao longo do livro, já que a recursividade é uma característica comum em programas funcionais.

2.4 TIPOS AGREGADOS

Podemos fazer coisas mais interessantes com tipos agregados ou

compostos, que são formados pela composição de tipos simples. Um par pode ser criado colocando dois valores dentro de parênteses, separados por uma vírgula:

```
 # let p = (1, "ichi");;
val p : int * bytes = (1, "ichi")
```

O tipo do par `p` é mostrado como `int * bytes`, que é o tipo dos pares formados por um `int` e uma string. O primeiro componente de um par pode ser acessado usando a função `fst`, e o segundo com `snd`:

```
 # fst p;;
- : int = 1

# snd p;;
- : bytes = "ichi"
```

Mas em geral, é mais prático obter os componentes de um par usando um `let`. No exemplo a seguir, criamos simultaneamente duas novas variáveis (`n` e `str`) para guardar os componentes do par:

```
# let n, str = p;;
val n : int = 1
val str : bytes = "ichi"
```

Isso é um exemplo do uso do *pattern matching* (*casamento de padrões*) na criação de variáveis. O uso do *pattern matching* permite acessar componentes de estruturas complexas de uma maneira simples, e é geralmente considerada uma das características mais úteis das linguagens ML. Veremos mais detalhes sobre *pattern matching* no capítulo *Registros e variantes*, além de vários exemplos do seu uso ao longo do livro.

Tuplas

De forma similar aos pares, podemos trabalhar facilmente com *tuplas* com mais de dois componentes, como triplas ou quádruplas:

```
# let tr = (3.14, "pi", true);;
val tr : float * bytes * bool = (3.14, "pi", true)

# let qd = (2, "fast", 2.0, "furious");;
val qd : int * bytes * float * bytes = (2, "fast", 2., "furious")
```

A única forma de acessar os componentes de uma tupla de tamanho maior que 2 é usando *pattern matching*; as funções `fst` e `snd` só funcionam em pares, e não existem funções similares para outros tamanhos. Um fato interessante das tuplas (incluindo pares) é que os componentes podem ser de tipos diferentes, como visto nos exemplos anteriores.

Tuplas são úteis para guardar, de maneira simples, um conjunto de valores simples que estão relacionados. Por exemplo, pontos em três dimensões são determinados por três coordenadas. Para calcular a distância euclidiana entre dois pontos, usamos uma fórmula derivada do teorema de Pitágoras. O exemplo a seguir usa *pattern matching* no próprio parâmetro da função para já separar os componentes dos dois pontos:

```
# let dist (x1, y1, z1) (x2, y2, z2) =
    sqrt ((x1 -. x2) ** 2.0 +. (y1 -. y2) ** 2.0 +.
          (z1 -. z2) ** 2.0);;
val dist : float * float * float ->
           float * float * float -> float = <fun>

# dist (0.0, 0.0, 0.0) (1.0, 2.0, 3.0);;
- : float = 3.74165738677394133
```

Listas

Tuplas funcionam bem para dados agregados que têm tamanho fixo e conhecido previamente, incluindo elementos que podem ter tipos distintos. Quando não se sabe o número de elementos que o programa precisa manipular, podem se usar as listas.

Listas são a estrutura de dados mais tradicional da programação funcional, sendo usadas desde a linguagem LISP original. LISP,

aliás, significava *LISt Processing*; hoje em dia, o nome da linguagem é simplesmente *Lisp*, que não é uma sigla.

Para criar uma lista em OCaml, deve-se colocar os elementos entre colchetes, separados por ponto e vírgula:

```
# [1; 2; 3; 4];;
- : int list = [1; 2; 3; 4]
```

O REPL identifica o tipo de `[1; 2; 3; 4]` como `int list`, uma lista de inteiros. O tamanho de uma lista pode ser obtido com a função `List.length`:

```
# List.length [1; 2; 3; 4];;
- : int = 4
```

`List.length` é o nome da função `length` (*comprimento*, em inglês), dentro do módulo `List`. Muitas das funções da biblioteca padrão da linguagem OCaml são separadas em diferentes módulos; veremos como definir módulos no capítulo *Módulos*.

Listas podem ser concatenadas com o operador `@`:

```
# [1; 2; 3; 4] @ [5; 6; 7; 8];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

O uso de ponto e vírgula para separar os elementos em listas, em vez de vírgulas, é para diferenciar das tuplas. OCaml aceita, na maioria dos contextos, que uma tupla seja escrita sem os parênteses:

```
# 4, 5;;
- : int * int = (4, 5)
```

Dessa forma, a vírgula é usada pelo compilador para detectar tuplas que não são separadas por parênteses. Por isso, uma lista como `[1, 2, 3]` não é uma lista de inteiros, pois é interpretada pelo compilador como `[(1, 2, 3)]`, que é uma lista de tuplas `int * int * int` (com apenas um elemento):

```
# [1, 2, 3];;
- : (int * int * int) list = [(1, 2, 3)]
```

Além disso, listas em OCaml são coleções homogêneas, ou seja, todos os elementos devem ser do mesmo tipo. Uma lista com elementos de tipos diferentes é um erro de tipo para o compilador:

```
# [1; 2.3; 4];;
Error: This expression has type float but an expression
was expected of type int
```

É possível acessar um elemento específico de uma lista usando a função `List.nth`, especificando a lista e um índice (que começa em 0):

```
# let l1 = [0; 1; 2; 3; 4];;
val l1 : int list = [0; 1; 2; 3; 4]

# List.nth l1 2;;
- : int = 2
```

Mas na prática essa função é raramente usada, e sua eficiência pode deixar a desejar (a complexidade para acessar um elemento de índice i é proporcional a i). Existem muitas outras operações predefinidas para listas; por exemplo, podemos ordenar uma lista usando `List.sort`:

```
# List.sort compare [5; 9; 1; 7; 13; 21];;
- : int list = [1; 5; 7; 9; 13; 21]
```

`List.sort` possibilita o uso de diferentes funções de comparação, e `compare` é uma função de comparação genérica disponibilizada pela linguagem OCaml. No caso de inteiros, usar `compare` para ordenação coloca os números em ordem crescente, como visto no exemplo. Esse exemplo ficará mais claro quando virmos as funções de alta ordem no capítulo *Programação funcional*.

A função `List.filter` serve para *filtrar* os elementos de uma lista de acordo com algum critério. Dada uma lista `l` e um predicado, `List.filter` retorna uma lista com os elementos de `l` que satisfazem o predicado. Por exemplo, usando a função `par` definida anteriormente, podemos filtrar os números de uma lista

que são pares:

```
# List.filter par [3; 9; 2; 1; 12; 7; 8];;
- : int list = [2; 12; 8]
```

Essa foi apenas um primeiro contato com as listas. O verdadeiro poder das listas fica claro quando se usam as técnicas de programação funcional que veremos no capítulo *Programação funcional*. No capítulo *Registros e variantes*, veremos como podemos definir as listas ou coleções similares.

Declaração explícita de tipos

Em todos os exemplos até agora, deixamos o REPL inferir os tipos das variáveis ou funções definidas. Isso reflete a prática normal da programação OCaml, que é confiar na inferência de tipos do compilador.

Entretanto, às vezes, pode ser necessário ou desejável declarar explicitamente o tipo de uma variável, parâmetro ou função. Nesse caso, podemos usar a sintaxe `id : tipo`, em que `id` é o nome do identificador.

```
# let x : int = 3;;
val x : int = 3
```

Para declarar o tipo de um parâmetro de função, usa-se a mesma forma, mas é preciso utilizar parênteses:

```
# let quadrado (x : int) = x * x;;
val quadrado : int -> int = <fun>
```

É possível declarar o tipo de alguns parâmetros, e deixar o tipo dos outros para o sistema de inferência:

```
# let mult x (y : int) = x * y;;
val mult : int -> int -> int = <fun>
```

Finalmente, para declarar o tipo de retorno de uma função, usa-se `: tipo` após todos os parâmetros. Assim, a função `quadrado`

com todos os tipos declarados fica:

```
# let quadrado (x : int) : int = x * x;;
val quadrado : int -> int = <fun>
```

Porém, declarar os tipos explicitamente é dificilmente necessário, e é raramente feito na prática. Programadores OCaml tendem a confiar na inferência de tipos para definições locais, e declarar tipos apenas nas interfaces de módulos, como veremos no capítulo *Módulos*.

Os exemplos deste capítulo servem como um primeiro contato com a linguagem OCaml. Mas, para ver programas mais interessantes, precisamos aprender mais sobre a linguagem e sobre programação funcional. O capítulo a seguir mostra como criar tipos e combiná-los, e o capítulo *Programação funcional* mostra as técnicas da programação funcional que fundamentam o poder da linguagem OCaml.

Capítulo 3

REGISTROS E VARIANTES

Até agora, temos visto o uso de tipos predefinidos na biblioteca padrão da linguagem OCaml. Neste capítulo, veremos como criar novos tipos, começando com sinônimos de tipos e prosseguindo para tipos mais interessantes, como variantes (ADTs) e registros.

3.1 SINÔNIMOS DE TIPOS

Os tipos mais simples que podem ser criados pelo programador em OCaml são os sinônimos de tipo, que são apenas um nome diferente para um tipo já existente. O uso de um nome diferente pode ajudar na documentação, descrevendo melhor como um valor deve ser usado. Por exemplo, em um programa que manipula figuras no plano, podemos definir o tipo de um ponto em duas dimensões:

```
# type ponto2d = float * float;;
type ponto2d = float * float
```

Tipos são criados usando a palavra-chave `type`, e o REPL responde mostrando o novo tipo criado. Os nomes dos tipos também devem começar com uma letra minúscula ou *underscore*. A partir desse ponto, podemos usar o tipo `ponto2d`, por exemplo, em uma função que calcula a distância do ponto até a origem:

```
# let dist_origem (p : ponto2d) =
    sqrt ((fst p) ** 2.0 +. (snd p) ** 2.0);;
val dist_origem : ponto2d -> float = <fun>
```

Nesse caso, é necessário declarar o tipo de `p` explicitamente. Caso contrário, a inferência de tipos vai considerar que `p` é um par de `float` s:

```
# let dist_origem p =
    sqrt ((fst p) ** 2.0 +. (snd p) ** 2.0);;
val dist_origem : float * float -> float = <fun>
```

De fato, os tipos `point2d` e `float * float` são sinônimos, e um pode ser usado em qualquer contexto em que o outro também pode. Isso significa que, embora criar um sinônimo de tipo possa ajudar na documentação do programa, é uma documentação apenas para quem vai ler o programa.

Um tema constante na programação funcional com tipagem estática é procurar usar o sistema de tipos a nosso favor, fazendo com que o compilador verifique as restrições do projeto do programa e evite que os dados sejam usados de maneira errada. Imagine agora que queremos definir um tipo para retângulos. Podemos definir um retângulo usando quatro números:

```
# type retangulo = float * float * float * float;;
type retangulo = float * float * float * float
```

Mas esse tipo diz muito pouco sobre como um retângulo é representado. Os quatro números representam o canto superior esquerdo, a largura e a altura do retângulo? Ou as coordenadas do canto superior esquerdo e as do canto inferior direito? Ou talvez as coordenadas dos cantos superior direito e inferior esquerdo?

Tuplas são interessantes para algumas situações com dados agregados, e sinônimos de tipos podem ajudar a documentar o uso dessas tuplas. Entretanto, essa não é a melhor forma de organizar os tipos em um programa. Uma maneira mais interessante de fazer isso é usando registros.

3.2 REGISTROS

Registros em OCaml são similares aos de linguagens estruturadas como Pascal e aos `struct` s da linguagem C. Um registro é um agregado de valores que são tratados como uma unidade, assim como tuplas, mas nos registros os componentes são nomeados.

Para criar um tipo de registro, usamos a palavra-chave `type` e sua definição entre chaves, declarando cada componente e seu tipo (os componentes também são chamados de *campos* do registro). Voltando ao exemplo do uso de pontos, podemos definir pontos em duas dimensões como um registro em vez de uma tupla:

```
# type ponto2d = { x : float; y : float };;
type ponto2d = { x : float; y : float; }
```

Para criar um registro do tipo `ponto2d` , basta especificar os valores de cada campo, dentro de colchetes:

```
# { x = 2.0; y = 3.0 };;
- : ponto2d = {x = 2.; y = 3.}
```

O compilador inferiu automaticamente o tipo `ponto2d` para o registro `{ x = 2.0; y = 3.0 }` . A inferência nesse caso é feita pelo nome dos campos do registro. Também é possível criar um *construtor* para o registro (uma função que cria novas instâncias):

```
# let criar_ponto2d xp yp = { x = xp; y = yp };;
val criar_ponto2d : float -> float -> ponto2d = <fun>
```

No construtor, não é preciso que os parâmetros tenham nomes diferentes dos campos do registro, já que campos não são variáveis, portanto, não há conflito de nomes:

```
# let criar_ponto2d x y = { x = x; y = y };;
val criar_ponto2d : float -> float -> ponto2d = <fun>
```

Podemos simplificar a notação mais um pouco: a linguagem

OCaml permite que se defina o valor do campo `x` usando uma variável de nome `x`, sem precisar fazer `x = x` (e o mesmo para qualquer variável que tenha o mesmo nome de um campo):

```
# let criar_ponto2d x y = { x; y };;
val criar_ponto2d : float -> float -> ponto2d = <fun>
```

Para acessar os campos de um registro, usamos a familiar notação `reg.campo`, em que `reg` é a variável que tem valor de registro, e `campo` é o nome do campo:

```
# let p = criar_ponto2d 1.2 3.6;;
val p : ponto2d = {x = 1.2; y = 3.6}

# p.x;;
- : float = 1.2

# p.y;;
- : float = 3.6
```

Nomes de campos

Os nomes de campos em OCaml seguem as convenções léxicas para identificadores de variáveis (começar com letra minúscula ou *underscore*, seguido por zero ou mais letras, dígitos ou *underscore*s).

O fato de o compilador inferir automaticamente o tipo do registro por meio apenas do nome dos campos significa que o programador não precisa ficar repetindo o nome do registro no código. Por outro lado, um problema dessa abordagem é que ter dois ou mais registros com campos de mesmo nome gera ambiguidade para o compilador.

Por exemplo, para definir uma função que soma as coordenadas de um `ponto2d`, após ter apenas o tipo `ponto2d` definido, é simples:

```
# let sum_xy p = p.x +. p.y;;
val sum_xy : ponto2d -> float = <fun>
```

Mas talvez nossa aplicação precise lidar com pontos em três dimensões também, logo definimos um novo tipo de registro:

```
# type ponto3d = { x : float; y : float; z : float };;
type ponto3d = { x : float; y : float; z : float; }
```

Agora, se tentarmos criar uma função similar a `sum_xy`, mas para multiplicar as coordenadas de um ponto, o resultado pode não ser o esperado:

```
# let mult_xy p = p.x *. p.y;;
val mult_xy : ponto3d -> float = <fun>
```

O que acontece neste exemplo demonstra que o compilador OCaml (na versão 4.01) decide a ambiguidade preferindo o tipo mais recente, que tem campos com os nomes usados (no caso, `x` e `y`). O fato de o campo `z` não ser utilizado não é suficiente para o compilador decidir que a função deveria funcionar sobre `ponto2d` e não `ponto3d`.

Como `x` e `y` são campos presentes no tipo `ponto3d`, e esse foi o último tipo a usar campos com esses nomes, o tipo inferido para o parâmetro da função é `ponto3d` e não `ponto2d`. Isso pode ser resolvido declarando explicitamente o tipo do parâmetro:

```
# let mult_xy (p : ponto2d) = p.x *. p.y;;
val mult_xy : ponto2d -> float = <fun>
```

Outras técnicas podem ser usadas para lidar com as ambiguidades de termos dois ou mais registros com campos de mesmo nome, mas na prática isso raramente é necessário. Em vez de lidar com a ambiguidade, é melhor evitá-la completamente: programadores OCaml tendem a colocar registros como `ponto2d` e `ponto3d` em módulos diferentes, o que evita o conflito de nomes. O uso de módulos é o tema do capítulo *Módulos*.

3.3 VARIANTES SIMPLES

Os *tipos variantes* ou *tipos de dados algébricos* (muitas vezes referidos pela sigla *ADTs*, do inglês *Algebraic Data Types*) são uma característica importante do sistema de tipos da linguagem OCaml. Diferente dos registros, que são encontrados de uma forma ou outra em quase todas as linguagens atuais, os ADTs não possuem contrapartida exata nas linguagens mais conhecidas.

Entretanto, com a tendência atual de várias linguagens adotarem características das linguagens funcionais (como discutido no capítulo *Introdução*), muitas linguagens mais recentes (Scala, Rust, Hack, Swift) possuem construções que funcionam exatamente como os ADTs em OCaml.

Fazendo uma analogia aproximada, os tipos variantes em OCaml funcionam como uma mistura das `enum`s de Java e C/C++ com os tipos de união (`union`) em C, mas funcionando de maneira muito mais prática do que essa analogia pode dar a entender.

O uso mais simples de tipos variantes é similar a uma `enum`. Imagine que estamos programando um jogo que envolve partidas de *Jokenpo*, também conhecido como *Pedra, Papel e Tesoura*. Nesse jogo, para duas pessoas, cada jogador escolhe uma das três possibilidades: pedra, papel, ou tesoura, e faz um sinal correspondente com a mão. O vencedor é decidido através de regras simples:

- Pedra ganha de tesoura;
- Tesoura ganha de papel;
- Papel ganha de pedra.

Vamos definir um tipo variante simples para representar a escolha de cada jogador, que chamaremos de *mão*:

```
# type mao = Pedra | Papel | Tesoura;;
type mao = Pedra | Papel | Tesoura
```

Mais uma vez, usamos o `type`, porém agora o tipo só tem três valores, que são as alternativas `Pedra`, `Papel` e `Tesoura`. Estes são chamados de *construtores* do tipo `mao`, pois permitem criar um valor desse tipo. `Pedra`, `Papel` e `Tesoura` também são chamados de *variantes* do tipo `mao`, porque representam as duas possibilidades para valores desse tipo. Podemos usar os construtores normalmente como valores:

```
# Pedra;;
- : mao = Pedra

# let m1 = Tesoura;;
val m1 : mao = Tesoura
```

Os construtores devem ter nomes começando com letra maiúscula, seguida por zero ou mais letras (maiúsculas ou minúsculas), dígitos e *underscore*s. Isso explica porque os nomes de variáveis e funções devem começar com letra minúscula: para diferenciá-los dos construtores.

O TIPO BOOLEANO COMO VARIANTE

Exceto pelo nome dos construtores, o tipo predefinido `bool` poderia ser definido como um tipo variante:

```
(* definicao hipotetica *)
type bool = true | false
```

Essa definição não funciona em OCaml, pois os construtores `true` e `false` não começam com letra maiúscula. Além desse fato, o tipo `bool` é especial em OCaml por causa da expressão condicional e da avaliação de curto-circuito em expressões booleanas.

Pattern matching simples

Como usar um valor de um tipo variante? Para a forma simples que vimos até agora, a única coisa a fazer com um valor de ADT é testá-lo para saber a que variante o valor pertence. Nesse caso, poderíamos fazer o que normalmente se faz em uma linguagem sem *pattern matching* e usar um `if`. Vamos escrever uma função para determinar que mão venceria a mão do jogador (por exemplo, para criar uma IA que sempre ganha):

```
# let vence_de m =
    if m = Pedra then Papel
    else if m = Papel then Tesoura
    else Pedra;; (* m = Tesoura *)
val vence_de : mao -> mao = <fun>

# vence_de Tesoura;;
- : mao = Pedra
```

Isso funciona, mas não é idiomático em OCaml, e por boas razões: existe uma solução melhor. Esta seria o *pattern matching* da linguagem OCaml, que já vimos uma forma simples junto com o `let` no capítulo *Tipos e valores básicos*.

A forma mais usada do *pattern matching* em OCaml é a construção `match ... with`. A função `vence_de` anterior ficaria da seguinte forma usando *pattern matching*:

```
# let vence_de m =
    match m with
      Pedra -> Papel
    | Papel -> Tesoura
    | Tesoura -> Pedra;;
val vence_de : mao -> mao = <fun>
```

Mesmo essa forma mais simples do `match` já é melhor do que usar um `if`, por dois motivos: o primeiro é que a função é mais legível, mostrando claramente qual o resultado associado a cada variante do tipo `mao`. O segundo é que o uso do `match` indica ao compilador que um valor do tipo `mao` está sendo processado, e o compilador pode detectar erros nesse processamento.

Por exemplo, digamos que, muito tempo depois de escrever a função `vence_de`, decidimos escrever uma função `perde_de` que determina a mão perdedora em relação à outra mão. Como se passou muito tempo, não lembramos mais das variantes do tipo e vamos processar só dois casos (eu sei que nesse caso seria difícil esquecer, mas me acompanhe). Usando `if`, a função ficaria:

```
# let perde_de_errado1 m =
    if m = Pedra then Tesoura
    else Pedra (* m = Papel *);;
val perde_de_errado1 : mao -> mao = <fun>
```

Nessa função, assumimos (erroneamente) que, se a mão `m` passada como parâmetro não for `Pedra`, só sobra a possibilidade de ser `Papel`. Essa função vai retornar um valor errado para `m` igual a `Tesoura`:

```
# perde_de_errado1 Tesoura;;
- : mao = Pedra
```

Mas `Pedra` ganha de `Tesoura`. Esse tipo de erro no código fica ainda mais fácil de ocorrer quando o tipo processado pela função é refatorado após um tempo, criando novas alternativas ou alterando algumas já existentes.

Por outro lado, com o uso do `match`, o compilador pode avisar convenientemente que estamos esquecendo um caso:

```
# let perde_de_errado2 m =
    match m with
      Pedra -> Tesoura
    | Papel -> Pedra;;
Characters 26-74:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Tesoura
val perde_de_errado2 : mao -> mao = <fun>
```

O compilador OCaml aceita a função e define-a, entretanto, dá um aviso: o `match` deixa escapar um caso, e ainda diz qual caso está escapando: `Tesoura`. Também é possível tornar esse aviso do

compilador em um erro de compilação e impedir que a função seja definida, o que é recomendável para código em produção.

A forma geral do `match ... with` é:

```
match e with p1 -> e1 | p2 -> e2 | ... | pn -> en
```

Os nomes começando com `e` são expressões, e os começando com `p` são padrões. O valor da expressão `e` é determinado e casado (*matched*) com os padrões `p1`, `p2` etc., na sequência. O resultado do `match` é o valor da expressão associada ao primeiro padrão que casa com `e`. Em um `match`, todas as expressões de resultado (`e1`, `e2`, etc.) devem ter o mesmo tipo. Veremos mais detalhes sobre como podem ser os padrões em breve.

Também pode-se usar uma barra vertical antes do primeiro padrão, o que torna a sintaxe de cada braço do `match` uniforme, principalmente quando cada padrão começa uma nova linha:

```
match e with
| p1 -> e1
| p2 -> e2
...
| pn -> en
```

Além de resultar em um código mais legível do que usar uma série de expressões condicionais (`if`), o `match` também gera código mais eficiente, sendo compilado para uma tabela de saltos quando possível, e fazendo apenas os testes necessários quando não é possível. É normal em programas OCaml ver o `match` sendo usado o tempo todo, muito mais do que o `if`.

O tipo determina o algoritmo

Mesmo nesse exemplo simples do pedra, papel e tesoura podemos ver um dos princípios importantes da programação funcional com tipos: a estrutura do tipo que deve ser processado por uma função determina a estrutura da própria função. O tipo `mao`

foi definido da seguinte forma:

```
type mao = Pedra | Papel | Tesoura
```

Ou seja, *uma mão pode representar três casos: pedra, papel ou tesoura*. Essa estrutura do tipo `mao` é espelhada na estrutura da função `vence_de`:

```
let vence_de m =
  match m with Pedra -> ... | Papel -> ... | Tesoura -> ...
```

Pelo tipo de `m`, é possível saber que seu valor só tem três possibilidades e, portanto, para dar a resposta de qual mão vence `m` é só determinar a resposta para cada uma das três possibilidades. Essa é a forma geral de qualquer função que deve processar um valor do tipo `mao`.

Os tipos não são apenas para que o compilador encontre erros no programa, mas também servem para comunicar a intenção do criador de cada componente, e até ajudar na escrita do programa. Essa correspondência entre a estrutura de um tipo `T` e a estrutura das funções que processam valores do tipo `T` será um tema constante neste capítulo, assim como no resto do livro.

3.4 VARIANTES COM VALORES ASSOCIADOS

Os ADTs também podem incluir valores associados às variantes. Um caso típico é representar figuras geométricas (para um programa de desenho, por exemplo). Vamos tratar três tipos de figuras: retângulos, círculos e triângulos equiláteros.

Estamos mais interessados no tamanho de cada figura do que na posição. Assim, para o retângulo, precisamos guardar a largura e altura; para o círculo, apenas o raio; e para o triângulo (equilátero) guardamos o tamanho do lado. O tipo fica da seguinte forma:

```
# type figura = Retangulo of float * float | Circulo of float
```

```
              | Triangulo of float;;
type figura =
  Retangulo of float * float
| Circulo of float
| Triangulo of float
```

Uma variante como `Circulo of float` declara um construtor `Circulo` com um parâmetro, de tipo `float`. O uso de um construtor com um parâmetro é similar a uma chamada de função:

```
# Circulo 5.0;;
- : figura = Circulo 5.
```

Já o construtor `Retangulo of float * float` tem como parâmetros dois `float`s, mas como uma tupla (note que `float * float` é exatamente o tipo de uma tupla de dois `float`s). Assim, para construir um `Retangulo`, precisamos passar dois números em uma tupla:

```
# Retangulo (3.2, 1.2);;
- : figura = Retangulo (3.2, 1.2)
```

Pattern matching com valores

Continuando com a ideia de que o algoritmo para processar um valor segue a estrutura do tipo deste valor, podemos usar o tipo `figura` usando *pattern matching*. A estrutura do tipo `figura` é similar à do tipo `mao`, o que significa que a estrutura dos algoritmos para processar `figura`s é similar, mas agora temos de saber como acessar os valores associados a cada construtor.

Em um padrão, podemos ter não só construtores, mas também variáveis, que recebem os valores associados caso o padrão seja casado. Por exemplo, vamos escrever uma função que calcule o perímetro de uma figura. É um cálculo simples, em geral: para retângulos, somamos o dobro da largura com o dobro da altura; para triângulos equiláteros, é só multiplicar o lado por três; e para um círculo, temos de usar a fórmula da circunferência, que é igual

ao dobro do raio multiplicado por *pi*.

```
# let pi = 3.14159265359;;
val pi : float = 3.14159265359

# let perimetro f =
    match f with
      Retangulo (l, a) -> 2.0 *. l +. 2.0 *. a
    | Circulo r -> 2.0 *. pi *. r
    | Triangulo l -> 3.0 *. l;;
val perimetro : figura -> float = <fun>
```

O `match` na função `perimetro` funciona da seguinte forma: para um valor `f` do tipo `Figura`, verifique se o construtor de `f` foi `Retangulo`. Se foi, existem dois parâmetros `float` associados, chame-os de `l` e `a`, e retorne o valor da expressão `2.0 *. l +. 2.0 *. a`. Caso o construtor de `f` não seja `Retangulo`, teste se foi `Circulo`; em caso positivo, associe o valor do parâmetro de `Circulo` ao nome `r` e retorne o valor da expressão `2.0 *. pi *. r`. Se `f` foi construído com `Triangulo`, proceda de forma similar.

Ou seja, o match segue testando o construtor do valor no `match` até encontrar o correto, e depois atribui os valores associados aos nomes de variáveis que existem no padrão. Os nomes dessas variáveis são escolhidos pelo programador. A função `perimetro` funcionaria da mesma forma se fosse escrita assim:

```
# let perimetro f =
    match f with
      Retangulo (largura, altura) -> 2.0 *. largura +.
                                     2.0 *. altura
    | Circulo raio -> 2.0 *. pi *. raio
    | Triangulo lado -> 3.0 *. lado;;
val perimetro : figura -> float = <fun>
```

Se um construtor recebe `n` parâmetros, é um erro especificar menos de `n` variáveis em um padrão desse construtor. Mas às vezes, não estamos interessados no valor de um ou mais desses parâmetros.

Por exemplo, vamos pensar em uma função simples `redondo` que retorna `true` quando uma figura passada como parâmetro é redonda; e `false`, caso contrário. Para evitar discussões filosóficas sobre o que significa ser "redondo", definiremos que apenas o círculo é redondo. Nesse caso, não é necessário acessar os parâmetros dos construtores da figura, pois o tamanho da figura não influencia no fato de ele ser redondo ou não. Podemos dar nomes às variáveis mesmo assim:

```
# let redondo f =
    match f with
      Retangulo (l, a) -> false
    | Circulo r -> true
    | Triangulo l -> false;;
val redondo : figura -> bool = <fun>
```

Mas os nomes são inúteis, e os não utilizados podem ser sinal de erros no código (existe uma opção para o compilador que avisa quando variáveis em padrões não são usados). Para deixar claro que um valor deve ser ignorado, pode-se usar o nome especial `_` (apenas um *underscore*):

```
# let redondo f =
    match f with
      Retangulo (_, _) -> false
    | Circulo _ -> true
    | Triangulo _ -> false;;
val redondo : figura -> bool = <fun>
```

Na verdade, quando um construtor tem mais de um parâmetro, podemos ignorar todos os argumentos de uma vez, usando apenas um *underscore*:

```
# let redondo f =
    match f with
      Retangulo _ -> false
    | Circulo _ -> true
    | Triangulo _ -> false;;
val redondo : figura -> bool = <fun>
```

Podemos até usar um *underscore* para o padrão inteiro: isso

representa um padrão *coringa* que casa com qualquer valor do tipo correto:

```
# let redondo f =
    match f with
    | Circulo _ -> true
    | _ -> false;;
val redondo : figura -> bool = <fun>
```

Essa última forma da função `redondo` expressa exatamente a intenção: apenas valores construídos com a variante `Circulo` são redondos, e o resto (quaisquer que sejam) não é:

```
# redondo @@ Triangulo 5.23;;
- : bool = false
```

Uma desvantagem da última versão da função `redondo` é que, se a definição do tipo `figura` for alterada para incluir novas formas (por exemplo, elipses), a função `redondo` automaticamente responderá `false` para essas novas variantes, mesmo que isso seja incorreto. Na versão anterior da função, repetida adiante, isso não acontece:

```
let redondo f =
  match f with
    Retangulo _ -> false
  | Circulo _ -> true
  | Triangulo _ -> false
```

Nessa versão, se o tipo `figura` for aumentado com novas variantes, o compilador vai emitir um aviso informando que o `match` na função não é exaustivo, o que indica ao programador a necessidade de atualizar o seu código. Alguns programadores preferem não usar padrões coringa, pois consideram mais confiável ter o compilador verificando a exaustividade de cada `match` .

Entretanto, existem situações nas quais o número de casos a tratar é muito grande, e não usar um padrão coringa pode resultar em funções longas, mas triviais, que poderiam ser mais legíveis se usassem o padrão coringa. Como em outras questões ligadas à

programação, é preciso considerar as vantagens e desvantagens de usar esse tipo de padrão, e usá-los quando for vantajoso.

3.5 TIPOS RECURSIVOS

Um ADT pode apresentar recursividade em uma ou mais variantes, ou seja, uma variante do tipo pode ter um valor associado que é do próprio tipo. Um exemplo típico é o das listas. Embora a linguagem OCaml já tenha as listas predefinidas, vamos ver como seria para criar listas de números inteiros

Podemos pensar nas listas como um tipo de dados recursivo da seguinte forma: uma lista de inteiros pode ser:

1. vazia; ou
2. um *par* composto por um inteiro e outra lista (o *resto* da lista)

Isso já indica o uso de um tipo variante para listas, cuja definição ficaria:

```
# type lista_int = Nil | Cons of int * lista_int;;
type lista_int = Nil | Cons of int * lista_int
```

`Nil` representa a lista vazia, e `Cons` é o par de um inteiro e outra lista. Os nomes `Nil` e `Cons` são tradicionais: `Nil` tem a ver com *nulo*, e `Cons` está relacionado ao fato de que o par é a *construção* de uma nova lista a partir de um elemento e uma lista já existente. Para criar uma lista vazia, basta usar o construtor `Nil`, claro:

```
 # Nil;;
- : lista_int = Nil
```

A lista que contém apenas o número 1 pode ser formada pelo `Cons` do número 1 com a lista vazia:

```
# let l1 = Cons (1, Nil);;
val l1 : lista_int = Cons (1, Nil)
```

A lista que contém os números 2 e 1, nesta ordem, pode ser formada pelo `Cons` do número dois com a lista `l1`:

```
# let l2 = Cons (2, l1);;
val l2 : lista_int = Cons (2, Cons (1, Nil))
```

A lista `l2` poderia ser criada diretamente também: `l2 = Cons (2, Cons (1, Nil))` funcionaria da mesma forma que o exemplo anterior.

Pattern matching e a estrutura recursiva das listas

Mais uma vez, a estrutura das funções que processam listas segue a estrutura do tipo. Assim como os algoritmos recursivos e as provas por indução na matemática, o tipo `lista_int` tem dois casos: um caso *base* (ou seja, não recursivo) que é formado pela lista vazia; e um caso recursivo que adiciona um elemento (um inteiro) a uma outra lista. Da mesma forma, uma função que deve processar uma lista segue a mesma estrutura: para escrever uma função que calcula um resultado a partir de uma lista:

1. Determine o resultado da lista vazia (caso base).
2. Dado o resultado para uma lista `l` e um novo elemento `x`, determine como combinar o resultado para `l` com a presença do novo elemento `x`. Para obter o resultado da lista `l`, a própria função deve ser chamada recursivamente.

Por exemplo, uma função útil para listas é determinar o seu tamanho, que naturalmente chamaremos de `tamanho`. Usando o modelo que segue a estrutura da lista, temos:

1. O resultado de `tamanho` para a lista vazia (caso base) é obviamente zero.
2. Se uma lista é formada por um elemento `x` mais o resto (que é outra lista) `rl`, o tamanho total é o tamanho de `rl` mais um.

Isso leva ao seguinte código para a função `tamanho` (lembrando de que funções recursivas devem ser definidas com `let rec` , como visto no capítulo *Tipos e valores básicos*):

```
# let rec tamanho l =
    match l with
      Nil -> 0
    | Cons (x, rl) -> 1 + tamanho rl;;
val tamanho : lista_int -> int = <fun>
```

Os resultados da função `tamanho` são os esperados:

```
# tamanho Nil;;
- : int = 0

# tamanho (Cons (1, Nil));;
- : int = 1

# tamanho l2;;
- : int = 2
```

3.6 ÁRVORES

A estrutura geral de um valor com tipo variante é uma árvore. Os construtores definem diferentes tipos de nós, e os construtores que têm parâmetros recursivos (ou seja, que contêm parâmetros de mesmo tipo) são nós internos, e os que não apresentam recursividade são folhas. A estrutura das listas que vimos antes é a de uma árvore na qual todo nó tem um filho que é uma folha, e um outro que é outra árvore, como pode ser visto no exemplo da figura:

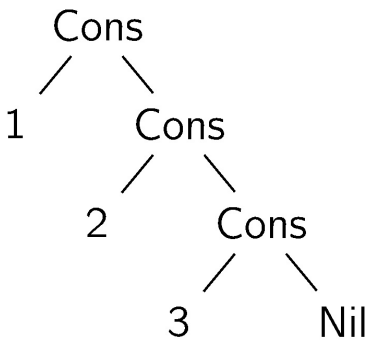

Figura 3.1: Estrutura da lista Cons (1, Cons (2, Cons (3, Nil))) como uma árvore

Portanto, o resultado do suporte avançado a tipos variantes e *pattern matching* é que aplicações que precisam trabalhar com estruturas de árvores são muito mais fáceis de escrever em uma linguagem funcional como OCaml do que nas linguagens imperativas. Muitas aplicações interessantes lidam com árvores: compiladores e outros processadores de linguagens, aplicações de processamento de linguagem natural, programas de otimização e resolução de problemas etc.

Vamos ver o exemplo de árvores binárias contendo números inteiros. A definição do tipo para árvores de inteiros é simples. Uma árvore pode ser apenas uma folha terminal, sem conteúdo, ou um nó contendo um valor inteiro e dois filhos, cada um podendo ser outra árvore. Essa é uma representação padrão para árvores binárias.

```
# type arvore_int =
  Folha
| No of arvore_int * int * arvore_int;;
type arvore_int = Folha | No of arvore_int * int * arvore_int
```

Exemplos de árvores binárias são mostrados na figura seguinte. Para representá-las em OCaml, podemos construir os valores aos poucos, aproveitando as partes em comum.

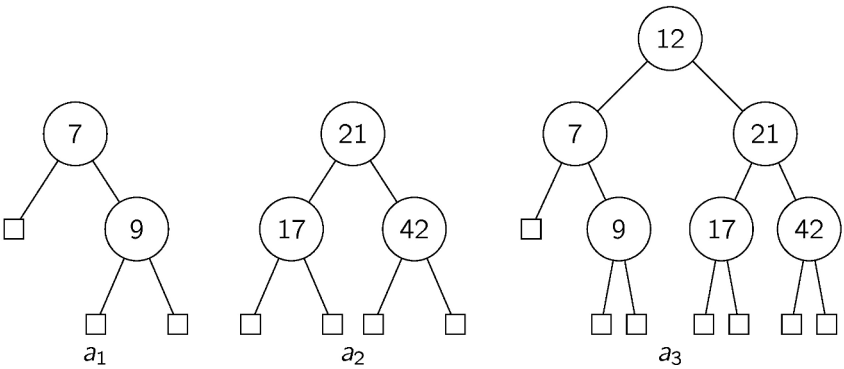

Figura 3.2: Exemplos de árvores binárias

```
# let a1 = No (Folha, 7, No (Folha, 9, Folha));;
val a1 : arvore_int = No (Folha, 7, No (Folha, 9, Folha))

# let a2 =
    No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha));;
val a2 : arvore_int =
  No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha))

# let a3 = No (a1, 12, a2);;
val a3 : arvore_int =
  No (No (Folha, 7, No (Folha, 9, Folha)), 12,
      No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha)))
```

Usando um tipo como `arvore_int`, podemos facilmente definir funções que processam árvores de inteiros, seguindo o princípio de fazer a estrutura do algoritmo seguir a estrutura do tipo. Se o objetivo for somar os números em uma árvore, uma função `soma_arvore` pode ser definida seguindo a estrutura do tipo: a soma de uma árvore vazia (uma `Folha`) é zero, e a soma de um `No (a1, n, a2)` é a soma das subárvores `a1` e `a2`, adicionada ao valor `n`:

```
# let rec soma_arvore a =
    match a with
      Folha -> 0
    | No (a1, n, a2) -> soma_arvore a1 + n + soma_arvore a2;;
val soma_arvore : arvore_int -> int = <fun>
```

Com essa função, podemos somar os valores das árvores do

exemplo anterior:

```
# soma_arvore a1;;
- : int = 16

# soma_arvore a2;;
- : int = 80

# soma_arvore a3;;
- : int = 108
```

Da mesma forma como definimos listas e árvores de inteiros, podemos definir listas de outros tipos de elementos, mas as definições seriam praticamente idênticas, variando apenas os tipos. O que queremos realmente é um tipo de listas que possa guardar elementos de qualquer tipo. Para isso, usamos a técnica de *polimorfismo*, que é o assunto do capítulo a seguir.

Capítulo 4

POLIMORFISMO E MAIS PADRÕES

Nos capítulos anteriores, vimos como definir novos tipos compostos a partir de combinações de tipos simples e constantes. Às vezes, queremos definir um tipo (ou uma função) que funcione com componentes variáveis.

O exemplo típico é uma coleção que possa conter elementos de qualquer tipo em vez de apenas tipos específicos como no capítulo anterior. Para ter coleções *genéricas*, que funcionam com vários tipos de elementos, é preciso empregar alguma forma de *polimorfismo*, que é o assunto deste capítulo. Começamos a examinar o uso de polimorfismo em OCaml a partir do exemplo das listas.

Listas polimórficas

O tipo `lista_int` , visto no capítulo anterior, representa listas de números inteiros. Seguindo o mesmo modelo, poderíamos facilmente definir listas de outros tipos, como listas de `float` s ou listas de strings. Entretanto, isso seria um trabalho repetido e que precisaria ser feito para cada novo tipo que quiséssemos usar em listas.

O que queremos, realmente, é dizer: *para qualquer tipo X, uma lista de X ou é vazia, ou é formada por um elemento do tipo X mais*

uma outra lista de X. Para isso, é preciso usar o *polimorfismo* da linguagem OCaml.

A palavra *polimorfismo* tem raiz grega e significa apenas "muitas formas". É uma palavra vaga com vários significados diferentes e não compatíveis. Programadores acostumados ao jargão da Orientação a Objetos normalmente associam o polimorfismo à possibilidade de usar um objeto declarado como uma classe B em contextos que esperam um objeto da classe A, desde que B seja uma classe derivada de A (direta ou indiretamente). Entre os pesquisadores em linguagens de programação, esse tipo de polimorfismo é conhecido como polimorfismo *ad hoc*.

O polimorfismo que ocorre em OCaml é diferente, chamado de *polimorfismo paramétrico*, e é mais similar aos *genéricos* de linguagens como C++, C# e Java, do que ao polimorfismo *ad hoc*. Queremos representar um tipo `lista` genérico que seja definido da forma: *para todo tipo T, uma lista de valores T ou é vazia, ou é um par de um valor do tipo T e uma lista de T*. Ou seja, o tipo `lista` vai precisar de um *parâmetro de tipo*, que é justamente T (daí o nome *polimorfismo paramétrico*).

Neste caso, T também é chamada de *variável de tipo*, pois pode representar vários tipos diferentes, assim como uma variável inteira `x` pode representar vários inteiros diferentes. Em Java ou C++, uma lista de valores de tipo T seria escrita como `Lista<T>` , em que T é uma variável de tipo.

Em OCaml, as variáveis de tipo seguem uma notação diferente: um nome de variável de tipo deve começar com um apóstrofo (*single quote*), e elas normalmente são representadas com letras minúsculas. O tipo *lista de valores do tipo* `'a` é escrito em OCaml como `'a lista` ; o parâmetro de tipo vem antes do nome (diferente da notação usada em Java, C++, C# e outras). A lista genérica em OCaml pode ser definida da seguinte forma:

```
# type 'a lista = Nil | Cons of 'a * 'a lista;;
type 'a lista = Nil | Cons of 'a * 'a lista
```

O nome do tipo é `'a lista`, pois, para uma lista específica, é preciso determinar o valor da variável de tipo `'a`. O construtor `Nil` representa o caso base e é idêntico à lista de inteiros vista anteriormente, já que não carrega nenhum valor. O construtor `Cons` representa a construção de uma lista pela adição de um elemento do tipo `'a` com uma lista de tipo `'a lista`.

> **Monomorfismo**
>
> Um tipo definido sem variáveis de tipo, como o `lista_int` visto no capítulo *Registros e variantes*, é chamado de tipo *monomórfico*, pois só possui uma forma. Isso é para contrastar com os tipos polimórficos como `'a lista`.

Felizmente, a inferência de tipos em OCaml ajuda a escrever valores de tipos polimórficos sem precisar especificar qual o valor da variável de tipo `'a`. A lista vazia continua sendo `Nil`, mas agora é polimórfica, afinal, é impossível determinar o tipo `'a`, já que a lista não possui nenhum elemento:

```
# Nil;;
- : 'a lista = Nil
```

Porém, assim que a lista contiver um ou mais elementos, a inferência de tipos identifica o valor de `'a`:

```
# Cons (1, Nil);;
- : int lista = Cons (1, Nil)

# Cons ("Arthur", Cons ("Dent", Nil));;
- : string lista = Cons ("Arthur", Cons ("Dent", Nil))
```

O tipo `int lista` representa uma lista de `int`s, o tipo

`string lista` representa listas de strings, e assim por diante. A convenção de colocar o parâmetro de tipo antes do seu nome funciona melhor em inglês: `int list` é exatamente a expressão que seria usada, em inglês, para *lista de inteiros*.

A função `tamanho`, para calcular o tamanho de uma lista polimórfica, tem exatamente o mesmo código da versão monomórfica:

```
# let rec tamanho l =
    match l with
    | Nil -> 0
    | Cons (_, rl) -> 1 + tamanho rl;;
val tamanho : 'a lista -> int = <fun>
```

Isso é uma demonstração da praticidade desse tipo de polimorfismo em OCaml.

4.1 AS LISTAS PREDEFINIDAS

Como vimos no capítulo *Tipos e valores básicos*, OCaml já inclui o tipo de listas predefinido. As listas predefinidas na linguagem são similares ao tipo `lista` definido anteriormente, com algumas conveniências sintáticas a mais.

O tipo das listas se chama `list`, com um parâmetro de tipo (portanto, `'a list`). O construtor `Nil` para listas vazias tem nome `[]`:

```
# [];;
- : 'a list = []
```

E o construtor `Cons` tem nome `::`, dessa forma, `Cons (1, Cons (2, Nil))` na notação que usamos antes é equivalente à seguinte expressão na notação predefinida:

```
 # 1 :: 2 :: [];;
- : int list = [1; 2]
```

Além disso, é possível escrever listas usando a notação `[x; y; z]` como um sinônimo para `x<:: y<:: z<:: []`. O compilador também imprime listas usando essa notação com elementos entre colchetes, separados por ponto e vírgula, como visto no exemplo anterior.

As mesmas convenções podem ser usadas para fazer `match` com listas. A função que calcula o tamanho de uma lista, usando as listas predefinidas, fica:

```
# let rec tamanho l =
    match l with
      [] -> 0
    | x :: rl -> 1 + tamanho rl;;
val tamanho : 'a list -> int = <fun>
```

Mas, como visto no capítulo *Tipos e valores básicos*, essa função já está definida na biblioteca padrão da linguagem como a função `List.length`:

```
# List.length ["Howard"; "Phillips"; "Lovecraft"];;
- : int = 3
```

Mais exemplos com listas

Dada uma lista de números inteiros, em muitas aplicações precisamos calcular a soma dos números presentes na lista. Lembrando do princípio de que a estrutura do tipo guia a estrutura do algoritmo, vemos que é preciso tratar dois casos: a lista vazia, e uma lista composta por um elemento e um resto de lista.

Para a lista vazia, podemos convencionar que a soma é zero, já que zero é o elemento neutro da soma. Se a lista é composta por um número e um resto da lista, o total é claramente a soma do número na frente da lista e a soma do resto, calculada recursivamente. Portanto, a função pode ser escrita da seguinte forma:

```
# let rec soma_lista l =
    match l with
```

```
      [] -> 0
    | x :: rl -> x + soma_lista rl;;
val soma_lista : int list -> int = <fun>
```

É interessante ver que, como foi usada uma operação de inteiros (a soma com `+`), o compilador infere que `soma_lista` recebe como parâmetro apenas listas de inteiros, e não outros tipos de listas. É fácil verificar que a função `soma_lista` funciona corretamente:

```
# soma_lista [];;
- : int = 0

# soma_lista [1; 3; 9; 21; 7];;
- : int = 41
```

Fazer uma função que segue a estrutura recursiva do tipo lista é relativamente direto, e evita completamente a programação com índices, algo que frequentemente causa erros de programação.

Outra situação frequente é precisar aplicar a mesma função a todos os elementos de uma lista, ou seja, *transformar* uma lista elemento a elemento. Para um exemplo mais específico, suponha que precisamos mudar a primeira letra de cada string em uma lista para maiúscula. Para uma string, apenas podemos usar a função `String.capitalize`:

```
# String.capitalize "slartibartfast";;
- : string = "Slartibartfast"
```

A tarefa é, dada uma lista de strings, passar para maiúscula o primeiro caractere de cada string, e formar uma lista com as strings transformadas. Conforme o processo para projeto de algoritmos que temos seguido, precisamos pensar nos dois casos para a estrutura de uma lista: no caso da lista vazia, o resultado da transformação é uma lista vazia também; se a lista for composta por um elemento na frente da lista (que é uma string) e um resto de lista, devemos criar uma lista que é composta pela transformação do elemento da frente (usando `String.capitalize`), seguida pela transformação do

resto da lista, chamando a própria função recursivamente. Desta forma, temos:

```
# let rec maiusculas_lista ls =
    match ls with
      [] -> []
    | s :: rls -> String.capitalize s :: maiusculas_lista rls;;
val maiusculas_lista : string list -> string list = <fun>
```

E a função funciona como esperado:

```
# maiusculas_lista ["joao"; "jose"; "da"; "silva"; "xavier"];;
- : string list = ["Joao"; "Jose"; "Da"; "Silva"; "Xavier"]
```

No capítulo a seguir, veremos como fazer esse tipo de transformação (ou *mapeamento*) de listas de maneira mais geral, usando funções de alta ordem.

Desconstruindo uma lista sem match

O processamento das listas é baseado na "desconstrução" recursiva da lista, tirando um elemento da lista de cada vez. Esse estilo de programação com listas vem da linguagem Lisp, mas em alguns dialetos mais conhecidos de Lisp (Common Lisp e Scheme) não é possível fazer *pattern matching*, pelo menos não por padrão.

O processamento de listas em Lisp usa duas funções para desconstruir uma lista nos seus componentes: um elemento do *início* da lista e o *resto* da lista, usando as funções `car` e `cdr`. Em OCaml, também é possível usar funções para desmembrar uma lista, e as funções se chamam `List.hd` e `List.tl` (os nomes vêm das palavras em inglês *head*, a cabeça da lista, e *tail*, a cauda da lista):

```
# List.hd [1; 2; 3];;
- : int = 1

# List.tl [1; 2; 3];;
- : int list = [2; 3]
```

Essas funções são menos úteis em OCaml, pois a maioria dos

casos pode ser tratada com `match`, porém, às vezes é necessário usá-las. Uma observação é que `List.hd` e `List.tl` podem causar exceções, pois uma lista vazia não tem primeiro elemento nem resto:

```
# List.hd [];;
Exception: (Failure hd)

# List.tl [];;
Exception: (Failure tl).
```

Essas funções são similares a fazer um `match` com uma lista considerando apenas um dos dois casos. Em um `match`, podemos considerar os dois casos ao mesmo tempo e evitamos o problema.

4.2 MAIS SOBRE PADRÕES

O uso de *pattern matching* é de grande importância nas linguagens funcionais com tipagem estática. Por isso, essas linguagens incluem várias características que tornam o uso do `match` mais poderoso e prático. Esse é o caso da linguagem OCaml, e nesta seção veremos mais possibilidades do uso do `match`. Não é preciso memorizar todos esses detalhes agora, mas é uma boa ideia saber o que é possível fazer com padrões em diferentes situações.

Nos exemplos anteriores deste capítulo, temos usado padrões como parte da construção `match`. Na verdade, padrões podem aparecer também na definição de variáveis e funções. Já vimos um exemplo disso no capítulo *Tipos e valores básicos*, quando usamos `let` para obter os componentes de um par. Por exemplo, se construirmos um par `p`:

```
# let p = (1, 2);;
val p : int * int = (1, 2)
```

Depois, podemos obter os componentes de `p` usando as funções `fst` e `snd`. Entretanto, também podemos obter os dois

ao mesmo tempo usando um `let` com um padrão:

```
# let (p1, p2) = p;;
val p1 : int = 1
val p2 : int = 2
```

Enquanto as funções `fst` e `snd` só podem ser usadas com pares, usar um `let` para obter os componentes de uma tupla funciona com tuplas de qualquer tamanho:

```
# let x, y, z, w = 1.2, 3.4, 0.75, 0.11;;
val x : float = 1.2
val y : float = 3.4
val z : float = 0.75
val w : float = 0.11
```

Nesse exemplo, parece que estamos usando um tipo especial de `let` para *atribuição paralela* (ou seja, atribuindo várias variáveis ao mesmo tempo). Porém, na verdade, isso é apenas um uso da criação de tuplas e do `let` com padrão.

O lado direito na atribuição de qualquer `let` pode ser um padrão. Em geral, é recomendável usar no `let` apenas os padrões chamados *irrefutáveis*, ou seja, que não possuem outras possibilidades. Padrões com tuplas sempre são irrefutáveis, mas também poderíamos usar um `let` com padrão para obter o primeiro elemento de uma lista:

```
# let h :: _ = [1; 2; 3; 4];;
Characters 4-10:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val h : int = 1
```

Uma lista pode ser vazia, o que poderia fazer o padrão `h <::< _` falhar (e é isso que o compilador avisa com a mensagem). Neste caso específico, a lista não é vazia, pois sabemos exatamente qual é a lista, mas isso também torna esse exemplo inútil. Em um caso real só haveria interesse em obter o primeiro elemento de uma lista

desconhecida, e nesse caso o `let` pode falhar, disparando uma exceção:

```
# let l1 = [];;
val l1 : 'a list = []

# let h :: _ = l1;;
Characters 4-10:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception: Match_failure
```

Por esse motivo, deve-se evitar usar padrões *refutáveis* em um `let`, assim como normalmente não é uma boa ideia usar `List.hd` e `List.tl`. Se uma variável pode ter várias formas, é melhor usar `match`. Porém, se a variável pode ter apenas uma forma (como no caso de uma tupla), é mais conciso usar um `let` com padrão. Obviamente, essa é uma recomendação geral e, em casos específicos, pode ser melhor ir contra ela.

Os parâmetros em uma definição de função também podem ser padrões, de preferência irrefutáveis. Se uma função recebe um par como parâmetro, em vez de usar um `match`:

```
# let soma_par p =
    match p with
      (x, y) -> x + y;;
val soma_par : int * int -> int = <fun>
```

Podemos usar um padrão diretamente no argumento da função:

```
# let soma_par (x, y) = x + y;;
val soma_par : int * int -> int = <fun>
```

Padrões complexos

Um `match` pode acessar não só o construtor mais externo, mas também vários níveis de construtores, se existirem. Isso torna o `match` uma ferramenta extremamente prática para acessar componentes de estruturas de dados.

Um exemplo: em uma aplicação de desenho, queremos representar pontos de referência para operações de desenho (ponto inicial de uma curva, centro de um círculo etc.). Um ponto de referência pode ser um ponto predefinido (uma *âncora*) na área de desenho, ou pode ser especificado pelas coordenadas x e y. Os pontos predefinidos são os quatro cantos da área retangular, e o centro. Em termos de tipos, definimos:

```
# type ancora = Centro | SupEsq | SupDir | InfEsq | InfDir;;
type ancora = Centro | SupEsq | SupDir | InfEsq | InfDir

# type ponto_ref = Ancora of ancora | Coord of int * int;;
type ponto_ref = Ancora of ancora | Coord of int * int
```

Agora digamos que precisamos de uma função que, dada uma lista de pontos de referência, calcule a distância de cada ponto até a origem. A ideia é uma transformação ou mapeamento de lista, como já visto. A estrutura geral da função é a seguinte:

```
let rec dist_ref ps =
  match ps with
    [] -> []
  | p :: rps -> ...
```

Essa estrutura foi a que usamos para todas as funções de listas até agora, e segue a estrutura do tipo lista. Neste caso, o elemento `p` ainda pode ter várias possibilidades, logo, poderíamos usar outro `match`:

```
let rec dist_ref ps =
  match ps with
    [] -> []
  | p :: rps ->
    match p with
      Ancora a -> ...
    | Coord c -> ...
```

Como `a` é um tipo de enumeração e `c` é uma tupla, ainda precisaríamos fazer mais alguns `match` (ou usar `fst` / `snd` com a tupla, ou um `let` com padrão). Dessa forma, rapidamente a

estrutura dessa função fica complicada e difícil de ler.

Em linguagens sem ADTs e *pattern matching*, a situação é ainda pior, mas felizmente o `match` em OCaml pode tratar de todos esses níveis ao mesmo tempo. Para auxiliar na escrita da função `dist_ref`, vamos definir uma função para calcular a distância de um ponto até a origem, e definir constantes para a largura e altura da área de desenho:

```
# let dist x y = sqrt ((float x) ** 2.0 +. (float y) ** 2.0);;
val dist : int -> int -> float = <fun>

# let largura = 640;;
val largura : int = 640

# let altura = 480;;
val altura : int = 480
```

Além disso, vamos usar a convenção de que a origem (o ponto de coordenadas 0,0) é o canto inferior esquerdo. Dessa forma, a função é

```
# let rec dist_refs ps =
    match ps with
      [] -> []
    | Ancora InfEsq :: rps -> 0.0 :: (dist_ref rps)
    | Ancora InfDir :: rps -> (dist largura 0) :: (dist_ref rps)
    | Ancora SupEsq :: rps -> (dist 0 altura) :: (dist_ref rps)
    | Ancora SupDir :: rps ->
      (dist largura altura) :: (dist_ref rps)
    | Ancora Centro :: rps ->
      (dist (largura/2) (altura/2)) :: (dist_ref rps)
    | Coord (x, y) :: rps -> (dist x y) :: (dist_ref rps);;
val dist_refs : ponto_ref list -> float list = <fun>
```

Cada caso é simples de analisar. O primeiro é que uma âncora no canto inferior esquerdo está exatamente na origem, portanto, sua distância para a origem é zero. Uma âncora `InfDir` está a uma distância da origem que é igual à largura da área de desenho, mas usamos a função `dist` para ficar uniforme. Assim seguem todos os casos, até o de um ponto de referência com coordenadas arbitrárias

(construtor `Coord`).

O `match` está desconstruindo três tipos: a lista, a enumeração `ancora` (associada ao construtor `Ancora`) e a tupla associada ao construtor `Coord`. A função é mais fácil de ler desta forma do que usando vários `match` (e muito mais fácil de ler do que se não usássemos ADTs e `match`).

Isso serve como exemplo de `match` com padrões complexos. Mas seria mais elegante escrever uma função que apenas calcula a distância de um ponto de referência até a origem:

```
# let dist_ref p =
    match p with
      Ancora InfEsq -> 0.0
    | Ancora InfDir -> dist largura 0
    | Ancora SupEsq -> dist 0 altura
    | Ancora SupDir -> dist largura altura
    | Ancora Centro -> dist (largura/2) (altura/2)
    | Coord (x, y) -> dist x y;;
val dist_ref : ponto_ref -> float = <fun>
```

E depois usá-la como base para construir `dist_refs`:

```
# let rec dist_refs ps =
    match ps with
      [] -> []
    | p :: rps -> dist_ref p :: (dist_refs ps);;
val dist_refs : ponto_ref list -> float list = <fun>
```

Com isso, voltamos a seguir a estrutura dos tipos. A função `dist_ref` segue a estrutura do tipo `ponto_ref`, e a função `dist_refs` segue a estrutura das listas, já que transforma uma lista de `ponto_ref`. A estrutura da função `dist_refs` é a mesma da função `maisculas_lista` que foi vista na seção *As listas predefinidas*. Não é uma coincidência: ambas fazem uma transformação ou mapeamento de uma lista para outra.

Esse padrão é bastante comum na programação funcional e, para evitar a repetição, seria melhor não ter que escrever várias

funções com mesma estrutura, mudando apenas a transformação aplicada aos elementos da lista. Para resolver isso, vamos usar uma função genérica chamada `List.map` que captura exatamente esta estrutura, e em cada caso dizemos apenas qual a transformação que será aplicada em cada elemento da lista. Para isso, é preciso passar a função de transformação como parâmetro. Esse tipo de manipulação com funções como valores de primeira classe da linguagem é típica da programação funcional, e é assunto do próximo capítulo.

Padrões condicionais

Com um `match`, podemos facilmente determinar a estrutura de um valor e acessar seus componentes, mas às vezes é interessante também verificar uma condição que não é apenas estrutural. Isso é possível através do uso da palavra-chave `when` em um braço de um `match`. Neste caso, dizemos que a condição é uma *guarda* do padrão.

Digamos que queremos classificar os dias de acordo com a temperatura: qualquer dia com temperatura máxima superior a 30 graus será classificado como um dia quente; se tiver temperatura mínima inferior a 15, será um dia frio; e será agradável se nenhuma das condições anteriores ocorrer.

Vamos definir uma enumeração para a classificação:

```
type dia = Quente | Frio | Agradavel
```

Agora vamos escrever uma função que recebe um par de números como entrada, representando as temperaturas mínima e máxima do dia (nesta ordem) e determina a classificação do dia segundo a ideia descrita anteriormente. Um `match` no parâmetro da função nos permite acessar as temperaturas mínima e máxima, e com isso poderíamos usar uma sequência de expressões

condicionais (`if`). Mas a função fica mais legível usando padrões condicionais com guardas:

```
# let classifica temps =
    match temps with
      (_, max) when max > 30.0 -> Quente
    | (min, _) when min < 15.0 -> Frio
    | _ -> Agradavel;;
val classifica : float * float -> dia = <fun>
```

O que acontece se o dia tiver mínima inferior a 15 graus e máxima superior a 30 graus (um dia no deserto talvez)? A especificação da classificação dos dias, dada em linguagem informal, não deixa claro qual deve ser a resposta. Na função `classifica` um dia dessa forma será sempre classificado como `Quente` , já que cada braço de um `match` é tentado na sequência:

```
# classifica (0.0, 42.0);;
- : dia = Quente
```

Padrões com alternativas e intervalos

Em algumas funções, podemos querer tratar vários padrões da mesma forma. Por exemplo, vamos definir um tipo para representar naipes de cartas de baralho:

```
type naipe = Copas | Espadas | Ouro | Paus
```

Existem situações em jogos de cartas nas quais o mais importante é saber se a carta é vermelha (se for de Copas ou Ouros) ou se é preta (Espadas ou Paus). Vamos escrever uma função que determina se uma carta é vermelha, dado o naipe:

```
# let vermelha n =
    match n with
    | Copas -> true
    | Ouro -> true
    | Espadas -> false
    | Paus -> false;;
val vermelha : naipe -> bool = <fun>
```

A função é pequena, mas temos uma repetição de casos que pode ser evitada usando padrões com alternativas:

```
let vermelha n =
  match n with
  | Copas | Ouro -> true
  | Espadas | Paus -> false
```

Padrões com alternativas são especialmente úteis para construtores sem parâmetros, como os do tipo `naipe`, porém também podem ser usados com construtores que tenham parâmetros. Neste caso, todos as alternativas de um mesmo padrão devem declarar as mesmas variáveis, e cada uma elas deve ter o mesmo tipo nas alternativas.

Para padrões alternativos usando caracteres, existe mais um atalho na linguagem: em vez de escrever `'a' | 'b' | 'c' ... | 'z'`, é possível escrever `'a'..'z'`. Por exemplo, uma função que retorna `true`, se o caractere de entrada for uma letra maiúscula:

```
let maiuscula c =
  match c with
    'A'..'Z' -> true
  | _ -> false
```

Infelizmente, esse atalho só funciona com caracteres, o que reduz a sua utilidade.

Nomeando partes de um padrão

Às vezes, pode ser útil dar um nome a um padrão ou a uma parte de um padrão. Isso pode ser feito com os chamados *as-patterns*. A forma geral de um caso com *as-pattern* é `p as v`, em que `p` é um padrão e `v` é um nome de variável. Para dar nomes a partes de um padrão `p`, é preciso usar parênteses em torno do `as`.

Dar nome ao valor que corresponde a um padrão inteiro pode ser útil quando o `match` é feito sobre um valor que não tem nome,

porque é o resultado de outra expressão. O exemplo aqui é de uma função que retorna o menor e o maior valor de uma lista de números inteiros (por simplicidade, se a lista for vazia, a função retorna um par de zeros).

Primeiro, vamos escrever uma função para obter o último elemento de uma lista:

```
# let rec ultimo l =
    match l with
      [] -> 0
    | x :: [] -> x
    | _ :: resto -> ultimo resto;;
val ultimo : int list -> int = <fun>
```

Se a lista está vazia, a função `ultimo` retorna zero, já que ela vai ser usada apenas na função para obter o menor e maior valores de uma lista de inteiros. Para obter esses valores, vamos ordenar a lista em ordem crescente; após isso, o menor elemento vai ser o primeiro da lista, e o maior será o último. Vamos escrever a seguinte função:

```
# let min_max_lista l =
    match List.sort compare l with
      [] -> (0, 0)
    | min :: resto -> (min, ultimo resto);;
val min_max_lista : int list -> int * int = <fun>
```

Aparentemente, ela funciona bem:

```
# min_max_lista [5; 1; 9; 11; 42; 12];;
- : int * int = (1, 42)
```

Mas a resposta ao chamarmos a função com uma lista de um elemento pode não ser a esperada:

```
# min_max_lista [42];;
- : int * int = (42, 0)
```

Faz mais sentido dizer que, para uma lista de um elemento apenas, o mesmo valor é o mínimo e o máximo da lista. O problema é que na função `min_max_lista` passamos o `resto` da lista

ordenada para a função `ultimo`, porém deveríamos passar a lista inteira.

Entretanto, essa lista não tem nome, pois é o resultado da chamada `List.sort compare l`. Uma forma de resolver isso é dando um nome para o valor que satisfaz o `match`:

```
# let min_max_lista l =
    match List.sort compare l with
      [] -> (0, 0)
    | min :: resto as l_ord -> (min, ultimo l_ord);;
val min_max_lista : int list -> int * int = <fun>
```

Nesta função, o padrão `min<:: resto as l_ord` faz com que chamamos o valor que satisfaz o padrão `min<:: resto` de `l_ord`. Em seguida, esse valor é passado para a função `ultimo`. Agora a resposta é a esperada:

```
# min_max_lista [42];;
- : int * int = (42, 42)
```

Existem outras soluções possíveis nesse caso, por exemplo, usar um `let .. in` para dar um nome à lista ordenada antes de fazer o `match`. Nomear valores que satisfazem todo ou parte de um padrão não é necessário com frequência, mas às vezes é a melhor forma de escrever uma função.

Note também que não é uma boa ideia retornar um valor válido como um par de zeros para o caso da lista vazia, pois essa seria a resposta para uma lista contendo apenas um valor zero.

```
# min_max_lista [0];;
- : int * int = (0, 0)
```

A forma correta de tratar a possibilidade de não ter resposta para certos valores de entrada seria usando exceções (assunto do capítulo *Características imperativas*) ou, de forma funcional pura, usando valores de tipos opcionais, como será visto na seção *Árvores polimórficas e valores opcionais*.

No exemplo anterior, o *as-pattern* deu um nome ao valor que bate com o padrão inteiro. Para dar nome ao valor que satisfaz apenas parte de um padrão, deve-se usar parênteses ao redor da parte nomeada. Usando isso, poderíamos escrever uma nova versão de `min_max_lista` que usa `ultimo` apenas se a lista tiver dois ou mais elementos:

```
let min_max_lista l =
  match List.sort compare l with
    [] -> (0, 0)
  | [x] -> (x, x)
  | min :: (_ :: rs as resto) -> (min, ultimo resto);;
```

O último padrão divide o resto da lista (após `min`) em `_::rs` apenas para garantir que a lista ordenada tem, pelo menos, dois elementos, mas precisa se referir a esse resto inteiro para chamar a função `ultimo`.

match paralelo

Um `match` *paralelo* consiste em testar dois valores ao mesmo tempo. Isso não é exatamente uma construção separada da linguagem OCaml, mas apenas o uso de um `match` com padrões complexos sobre tuplas. Voltando ao exemplo do jogo Pedra, Papel e Tesoura, cujo tipo repetimos aqui:

```
# type mao = Pedra | Papel | Tesoura;;
type mao = Pedra | Papel | Tesoura
```

Podemos querer escrever uma função que especifica o resultado do jogo: quem ganhou, ou se foi empate. Para especificar esse resultado, definimos um ADT simples de enumeração:

```
# type result_jogo = J1Vence | J2Vence | Empate;;
type result_jogo = J1Vence | J2Vence | Empate
```

`J1Vence` representa um jogo em que o primeiro jogador vence, e `J2Vence` representa a vitória do segundo jogador. Agora podemos escrever uma função `resultado` que recebe as mãos dos

dois jogadores e calcula o resultado do jogo. Neste caso, queremos testar as duas mãos ao mesmo tempo, e dizer o que acontece em cada combinação possível. Isso é fácil usando um `match` paralelo, que consiste em formar uma tupla com as duas mãos e fazer `match` na tupla:

```
# let resultado m1 m2 =
    match m1, m2 with
      _ when m1 = m2 -> Empate
    | Pedra, Papel -> J2Vence
    | Pedra, Tesoura -> J1Vence
    | Papel, Pedra -> J1Vence
    | Papel, Tesoura -> J2Vence
    | Tesoura, Pedra -> J2Vence
    | Tesoura, Papel -> J1Vence;;
val resultado : mao -> mao -> result_jogo = <fun>
```

Para evitar ter de escrever todos os casos nos quais ocorre a igualdade, o primeiro caso do `match` no exemplo usa uma guarda para detectar um empate. Os demais determinam quem vence de acordo com a combinação que ocorreu.

> **Limites da verificação em um match**
>
> O exemplo da função `resultado` omitiu parte da saída do REPL, porque, ao definir essa função, o compilador emite um aviso de `match` que pode não ser exaustivo. Isso é causado pelo uso da guarda no primeiro caso; o compilador não é esperto o suficiente para decidir que esse primeiro caso satisfaz todos os casos que não são listados (o que seria difícil de fazer, já que a guarda é uma expressão booleana arbitrária).

Um `match` paralelo pode ser utilizado com mais de dois valores, construindo tuplas com três ou mais componentes. O problema é que o número de casos no `match` tende a crescer

bastante, o que pode deixar o código confuso. Na maioria das situações é mais comum ver o uso de `match` paralelo com apenas dois valores.

4.3 ÁRVORES POLIMÓRFICAS E VALORES OPCIONAIS

Na seção *Árvores*, vimos como definir uma estrutura de árvore binária para inteiros. Assim como no caso das listas, podemos usar o polimorfismo para definir uma versão genérica das árvores binárias. O tipo é definido da mesma forma que para as árvores binárias contendo inteiros, entretanto, agora usamos uma variável de tipo em vez do fixo `int`:

```
# type 'a arvore = Folha | No of 'a arvore * 'a * 'a arvore;;
type 'a arvore = Folha | No of 'a arvore * 'a * 'a arvore
```

Com essa definição, podemos definir árvores de números inteiros, exatamente como antes (a inferência de tipos determina o tipo correto):

```
# let a1 = No (Folha, 7, No (Folha, 9, Folha));;
val a1 : int arvore = No (Folha, 7, No (Folha, 9, Folha))

# let a2 = No
    (No (Folha, 17, Folha), 21, No (Folha, 42, Folha));;
val a2 : int arvore =
  No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha))
```

Mas também podemos definir árvores de outros tipos:

```
# let a_ch = No (No (Folha, 'c', Folha), 'h', Folha);;
val a_ch : char arvore = No (No (Folha, c, Folha), h, Folha)
```

O código para calcular a soma dos valores em uma árvore de inteiros é o mesmo, mudando apenas o tipo inferido:

```
# let rec soma_arvore a =
    match a with
      Folha -> 0
```

```
    | No (a1, n, a2) -> soma_arvore a1 + n + soma_arvore a2;;
val soma_arvore : int arvore -> int = <fun>
```

Busca em árvores binárias

Uma árvore binária é uma árvore em que cada nó pode ter até dois filhos, e na qual os valores seguem uma ordem de acordo com a sua estrutura: dado um nó `N`, todos os descendentes do lado esquerdo de `N` têm valores menores que o valor de `N`, e todos os descendentes do lado direito são maiores que `N`. As árvores mostradas na seção *Árvores* são binárias.

Uma operação comum em uma árvore binária é a busca de valores. Como os valores nela seguem uma ordem refletida na sua estrutura, uma busca binária consegue encontrar um valor na árvore (ou determinar que ele não existe) examinando apenas uma fração dos nós armazenados nela. Sendo mais preciso, a busca binária tem complexidade logarítmica em relação ao número de nós da árvore.

O processo da busca binária é naturalmente recursivo: comparamos o valor buscado com o valor da raiz da árvore. Se for igual, o nó procurado é a raiz. Se o valor buscado for menor que o da raiz, a busca continua recursivamente na subárvore esquerda; se for maior, a busca continua na subárvore direita. Se em algum ponto chegarmos a uma folha da árvore, o valor buscado não existe nela.

Com essa ideia de algoritmo e o tipo definido para árvores, é fácil escrever uma função recursiva de busca binária. O problema é o que a função deve retornar quando o valor não for encontrado na árvore. Quando a função encontra o valor buscado, a função retorna o nó cujo valor foi encontrado; isso porque normalmente estamos interessados em outros valores armazenados no mesmo nó, o que não é o caso aqui. Quando o valor não é encontrado, existem algumas possibilidades; uma delas é retornar um valor especial no

caso de uma busca sem sucesso.

Em muitas linguagens, uma referência pode ser nula, e existe um valor especial `null` que representa isso. Uma busca binária que não encontra o valor buscado poderia retornar `null`. O problema é que tentar utilizar uma referência nula geralmente leva a exceções, o que pode causar problemas em programas que não verificam o valor retornado. Hoje em dia, o uso de referências nulas é considerado uma má prática na programação, e uma herança maldita originada de linguagens com ponteiros, como a linguagem C.

Nas linguagens funcionais com tipagem estática, existe uma convenção de usar os *tipos opcionais* para casos em que uma função pode ou não retornar um valor válido. Os tipos opcionais apresentam muitas vantagens em relação ao uso de valores especiais como `null` para representar erros.

Um tipo opcional representa duas possibilidades: nenhum valor, ou algum valor de algum outro tipo. Em OCaml, os tipos opcionais são predefinidos como o tipo `'a option`, com duas variantes. A definição do tipo é:

```
type 'a option = None | Some of 'a
```

Os construtores são `None` (*nenhum*, em inglês) e `Some` (*algum*, em inglês), e este último tem um argumento. A ideia, como indicado pelos nomes dos construtores, é usar `None` para representar a inexistência de um valor, e `Some x` para representar que existe um valor e ele é `x`. Podemos construir valores de tipos opcionais facilmente:

```
# None;;
- : 'a option = None

# Some 5;;
- : int option = Some 5
```

```
# Some "shawarma";;
- : string option = Some "shawarma"
```

Note que, para um valor `Some x`, o compilador infere facilmente o tipo através do tipo de `x`, mas um valor `None` sozinho vai ficar com tipo polimórfico. Em uma função que retorna um tipo opcional, o contexto vai fazer com que o valor `None` seja atribuído ao tipo correto.

Usando o tipo `option`, podemos finalmente escrever a função de busca em árvore binária:

```
# let rec busca a v =
    match a with
      Folha -> None
    | No (a1, n, a2) when n = v -> Some a
    | No (a1, n, _) when v < n -> busca a1 v
    | No (_, n, a2) -> busca a2 v;;
val busca : 'a arvore -> 'a -> 'a arvore option = <fun>
```

Uma coisa interessante nessa função é que o tipo inferido para ela é polimórfico: a função `busca` recebe um parâmetro de tipo `'a arvore` (a árvore onde fazer a busca) e um parâmetro de tipo `'a` (o valor a buscar na árvore), para um tipo genérico `'a`. Isso ocorre porque as comparações em OCaml (igualdade e "menor que") são polimórficas:

```
 # (=);;
- : 'a -> 'a -> bool = <fun>

# (<);;
- : 'a -> 'a -> bool = <fun>
```

A função de busca funciona como esperado:

```
# let a2 =
    No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha));;
val a2 : int arvore =
  No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha))

# busca a2 111;;
- : int arvore option = None
```

```
# busca a2 17;;
- : int arvore option = Some (No (Folha, 17, Folha))
```

A forma mais comum de usar um valor de tipo opcional é por meio do `match`. Um exemplo é uma função que retorna apenas um booleano indicando se o valor buscado existe ou não na árvore. Para isso, podemos usar a função `busca` já definida:

```
# let contem a v =
    match busca a v with
      None -> false
    | Some a2 -> true;;
val contem : 'a arvore -> 'a -> bool = <fun>
```

Nesse caso, nem foi preciso examinar o valor retornado pela busca (a árvore `a2`), mas se fosse, poderíamos usar as técnicas que já vimos para lidar com valores de tipos variantes. O tipo `option` é usado com frequência em OCaml e outras linguagens funcionais com tipos estáticos, e vamos ver vários outros exemplos de seu uso ao longo do livro.

Neste capítulo, vimos como definir tipos polimórficos e como utilizar os valores desses tipos, além de mais algumas possibilidades de uso do `match`. Todos os exemplos vistos até agora são programas funcionais puros, mas existem outras técnicas importantes na escrita de programas funcionais, e é interessante examinar em mais detalhe o que caracteriza a programação funcional. Esses são tópicos do próximo capítulo.

Capítulo 5

PROGRAMAÇÃO FUNCIONAL

Todos os exemplos que vimos até agora são puramente funcionais, mas foram apenas exemplos simples. É possível fazer muito mais sem precisar recorrer às características imperativas da linguagem, e isso é o que ocorre na prática: muitos programas OCaml contêm a maior parte do código funcional, tendo apenas uma pequena parte imperativa.

Para poder ir mais longe usando apenas programação funcional, precisamos aprender mais algumas técnicas do paradigma, e é isso que faremos neste capítulo.

5.1 A ESSÊNCIA DA PROGRAMAÇÃO FUNCIONAL

As ideias de programação imperativa e programação funcional (às vezes, chamada de *aplicativa* ou *denotativa*) convivem desde o início da programação de alto nível. No final dos anos 50, surgiram as versões originais da linguagem FORTRAN e da linguagem LISP.

FORTRAN foi criada para facilitar a criação de programas para processamento numérico, o tipo de software mais comum na época, era uma linguagem imperativa, bastante próxima ao nível da linguagem de máquina. LISP foi criada para processamento simbólico, e em sua versão original era puramente funcional.

A principal diferença da programação funcional em relação à imperativa é que costuma ser chamada de *transparência referencial*: cada parte do programa funcional sempre tem o mesmo resultado, independente do contexto onde ele se encontra. Isso é similar à ideia matemática de *função*: uma função matemática (como a função *seno*) sempre deve ter o mesmo resultado se a entrada for a mesma. Ou seja, não faz sentido que o seno de zero seja igual a zero em um dado momento, e igual a 1 em outro momento.

O principal componente das linguagens imperativas que atrapalha a transparência referencial são os chamados *efeitos colaterais*, ou simplesmente *efeitos*. Um efeito comum é a mutabilidade das variáveis, ou seja, a possibilidade de mudar o valor de variáveis durante a execução. Para ver isso, imagine uma função `f` em um programa, e `f` usa o valor de uma variável global para calcular seu resultado. Se o valor da variável global muda, o resultado de `f` muda, o que significa que uma função em um programa imperativo (com efeitos) não corresponde ao conceito matemático de função.

O fato de que um componente do programa (como uma função) pode ter resultados diferentes em contextos (e momentos) diferentes — ou seja, quando não há transparência referencial — dificulta a compreensão de cada componente, pois é preciso considerá-lo de maneira separada em cada possível contexto. Também limita a capacidade de composição do programador, porque cada combinação de componentes exige uma análise separada.

Desse ponto de vista, os efeitos são ruins. Mas, ao mesmo tempo, não podemos simplesmente evitar todos eles. Um programa totalmente sem efeitos pode até chegar a resultados, mas não pode imprimi-los na tela, já que operações de entrada e saída são também efeitos.

Por isso as linguagens de programação funcional tentam controlar ou limitar o uso dos efeitos. Em uma linguagem funcional pura, como Haskell, eles são controlados explicitamente através das *mônadas*, como uma forma de manter a transparência referencial. Já em linguagens como OCaml e Scheme, que não são puras, procura-se limitar o uso dos efeitos para ter os benefícios da programação funcional na maior parte do tempo, mas ter acesso a eles de forma prática quando necessário.

POR QUE SE IMPORTAR COM TRANSPARÊNCIA REFERENCIAL?

Para uma leitura interessante e com exemplos práticos sobre as vantagens de ter transparência referencial (pois aumenta a modularidade do programa e a capacidade de compor programas a partir de partes simples), é recomendada a leitura do artigo *Why Functional Programming Matters*, de John Hughes, disponível em http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

5.2 MUTABILIDADE E OUTROS EFEITOS

OCaml é uma linguagem funcional *híbrida*, que procura incentivar a programação funcional, mas deixa disponível para o programador o uso dos efeitos imperativos de maneira similar às linguagens imperativas. Um ponto em que a linguagem OCaml evita os efeitos é nas variáveis: todas as variáveis em OCaml são imutáveis e seu valor não pode ser mudado. É possível usar mutabilidade em OCaml (como será visto no capítulo *Características imperativas*), mas de forma não transparente e que exige um pouco mais de sintaxe.

Em geral, os programadores OCaml só usam mutabilidade quando é realmente necessário. E com as ferramentas da programação funcional, essa necessidade só acontece raramente.

Outros tipos de efeitos são mais comuns, por exemplo, operações de entrada e saída, e exceções. Qualquer programa não trivial precisa realizar entrada e saída, e as exceções são frequentemente usadas em OCaml para tratar de situações de erro.

5.3 PROGRAMAÇÃO RECURSIVA

Uma reação comum aos programadores com experiência em linguagens imperativas no primeiro contato com a programação funcional é perguntar "como é possível programar sem mudar os valores das variáveis?". A resposta é simples: é preciso mudar a forma de pensar as soluções para problemas de programação.

Um padrão comum na programação imperativa é processar uma coleção de itens em coleções usando vetores (*arrays*). A programação imperativa surgiu principalmente de aplicações numéricas e científicas, em que era preciso calcular a resposta baseado no processamento de vetores e matrizes de números.

Processar coleções (principalmente *arrays*) usando *loops* leva ao uso de variáveis mutáveis para armazenar os índices ou contadores do *loop*. Também é preciso usar variáveis mutáveis para acumular resultados a partir dos valores em uma coleção, por exemplo. Os *loops* são repetições de comandos imperativos, porém a combinação dos comandos é bastante dependente da ordem em que eles acontecem e dos efeitos envolvidos. Isso significa que, para entender um programa imperativo, é preciso "simular" mentalmente a sua execução, passo a passo, o que está sujeito a erros (a ocorrência frequente de erros nos limites dos índices em um *loop* é uma demonstração dessa dificuldade).

Na programação funcional, o padrão mais comum é usar funções recursivas em vez de *loops* com mutação. Funções recursivas seguem a estrutura do tipo e processam elementos sem usar índices, como vimos nos exemplos com listas no capítulo *Registros e variantes*.

Um exemplo é uma função para calcular a soma dos elementos em uma coleção. No caso imperativo, a coleção é normalmente um *array*, e a estrutura da função seria um *loop* com duas variáveis mutáveis: um contador/índice para o *loop* (para acessar elementos do array) e uma variável `soma` que acumula o valor da soma vista até o momento. No final, a função retornaria o valor da variável `soma`.

Uma versão funcional usaria uma lista em vez de *array*, como já vimos na função `soma_lista` no capítulo *Registros e variantes*:

```
# let rec soma_lista l =
    match l with
      [] -> 0
    | x :: rl -> x + soma_lista rl;;
val soma_lista : int list -> int = <fun>
```

Essa função não usa nenhuma variável mutável. Os elementos são acessados usando `match` dentro de uma função recursiva (que isola um elemento diferente da lista em cada chamada). O resultado final é acumulado como o resultado da chamada recursiva, sem precisar de uma variável mutável para isso.

O uso de funções recursivas pode trazer duas preocupações: uma com relação à compreensão das funções e outra com relação à sua eficiência. A preocupação com a compreensão das funções vem muitas vezes da falta de experiência dos programadores com programação recursiva. Então, surgem perguntas sobre como entender a execução de uma função recursiva (em termos puramente funcionais, isso é o mesmo que pensar na *avaliação* da expressão que é o corpo da função).

A preocupação pela eficiência vem da ideia de que funções recursivas são menos eficientes que usar *loops* imperativos, já que normalmente a implementação das chamadas de funções em uma linguagem de programação exige o uso de espaço na pilha e manipulação do contexto de execução. As respostas a essas preocupações envolvem princípios importantes da programação funcional, e serão examinadas a seguir.

Avaliação como substituição

Uma vantagem de programas puramente funcionais (com transparência referencial) é que é possível entender como funciona a avaliação de qualquer expressão de um programa de forma similar à avaliação de expressões algébricas na matemática: "simplificando" a expressão. O processo de simplificação de uma expressão consiste em substituir partes de uma expressão por partes equivalentes mais simples. Isso só é possível se toda subexpressão é *transparente*, ou seja, se não possui efeitos e pode ser livremente substituída por expressões equivalentes.

Vamos ver um exemplo da avaliação da função `soma_lista` com a lista `[1; 2; 3; 4]`. O primeiro passo é fazer um `match` da lista `[1; 2; 3; 4]`, o que casa com o segundo braço do `match`, que é `x :: rl`, associando `x` a `1` e `rl` a `[2; 3; 4]`. Assim, temos a seguinte equivalência (vamos usar o símbolo `=>` para mostrar a avaliação de uma expressão):

```
soma_lista [1; 2; 3; 4] => 1 + soma_lista [2; 3; 4]
```

A chamada recursiva é com o argumento `[2; 3; 4]`. A avaliação continua de forma similar:

```
soma_lista [1; 2; 3; 4]
  => 1 + soma_lista [2; 3; 4]
  => 1 + 2 + soma_lista [3; 4]
  => 1 + 2 + 3 + soma_lista [4]
  => 1 + 2 + 3 + 4 + soma_lista []
```

```
  => 1 + 2 + 3 + 4 + 0
```

No penúltimo passo, a chamada `soma_lista []` casa com o primeiro braço do `match`, resultando em 0. Dessa forma, a soma dos números da lista chega ao resultado observado:

```
# soma_lista [1; 2; 3; 4];;
- : int = 10
```

Iteração e recursão

Algumas pessoas ficam preocupadas ao descobrir que a programação funcional incentiva a escrita de funções recursivas. Talvez porque tenham ouvido recomendações de professores de programação para não usar recursividade por motivos de eficiência.

Pessoas que tenham conhecimento sobre a implementação de linguagens de programação podem associar a recursividade ao uso de uma pilha de tamanho fixo, como é feito em muitos sistemas de tempo de execução, e podem se perguntar o que acontece se uma função recursiva que processa uma lista recebe como parâmetro uma lista muito grande.

De fato, uma função recursiva em OCaml pode causar um estouro de pilha. A função a seguir demonstra a situação:

```
# let rec loop_infinito x =
    x + loop_infinito x;;
val loop_infinito : int -> int = <fun>

# loop_infinito 10;;
Stack overflow during evaluation (looping recursion?).
```

Isso ocorre de maneira similar em linguagens imperativas. Mas em uma linguagem funcional como OCaml, existem situações em que a recursividade pode ser implementada de maneira eficiente e usando espaço constante na pilha. Um exemplo é a seguinte função, que parece com o primeiro `loop_infinito`, mas se comporta de forma diferente:

```
# let rec loop_infinito2 x =
    loop_infinito2 x;;
val loop_infinito2 : 'a -> 'b = <fun>

# loop_infinito2 10;;
```

Essa última chamada `loop_infinito2 10` nunca termina, e o interpretador entra realmente em um *loop* infinito. A diferença entre `loop_infinito` e `loop_infinito2` é importante e reflete uma característica das linguagens funcionais: a ideia de que uma função recursiva não necessariamente representa um processo recursivo (ou seja, uma função recursiva não necessariamente precisa ser implementada com recursão).

Podemos ver isso usando a ideia de avaliação como substituição. A chamada `loop_infinito 10` pode ser avaliada da seguinte forma, usando a definição da função `loop_infinito`:

```
loop_infinito 10
  => 10 + loop_infinito 10
  => 10 + 10 + loop_infinito 10
  => 10 + 10 + 10 + loop_infinto 10
  => ...
```

Em cada passo, a chamada `loop_infinito 10` é substituída por `10 + loop_infinito 10`, de acordo com a definição. Em cada ponto da avaliação, existe uma chamada `loop_infinito 10` pendente, além de várias somas. Todo esse trabalho pendente deve ficar armazenado na memória de alguma forma, para que seja possível calcular o valor correto quando a função retornar.

O fato de que essa função específica nunca retorna não importa, o mecanismo de implementação é geral. Na implementação da linguagem OCaml, assim como em outras linguagens, uma pilha é utilizada para guardar o contexto das operações pendentes, e ela tem um limite de tamanho. Por isso, a função `loop_infinito 10` causa um estouro de pilha.

Enquanto isso, a avaliação da chamada `loop_infinito2 10`

ocorre da seguinte forma:

```
loop_infinito2 10
  => loop_infinito2 10
  => loop_infinito2 10
  => loop_infinito2 10
  => ...
```

Apesar de a função `loop_infinito2` também ser recursiva, podemos ver que a quantidade de trabalho que fica pendente entre as chamadas recursivas não aumenta. Não existe contexto a guardar para quando a função retornar (se retornar), e por isso `loop_infinito2` não precisa usar a pilha. De fato, ela é implementada exatamente como um *loop* infinito, sem chamadas de função.

A diferença entre `loop_infinito` e `loop_infinito2` é a forma como a chamada recursiva é feita. A função `loop_infinito2` apresenta um tipo de recursividade conhecido como *recursividade de cauda* (do inglês, *tail recursion*), o que significa que a chamada recursiva é a última coisa que a função faz. A `loop_infinito` não apresenta recursividade de cauda: após obter o resultado da chamada recursiva (se fosse possível obtê-lo), ainda é preciso somar o resultado com o valor de `x`, e portanto a chamada recursiva não é a última operação que a função deve executar.

Quando a chamada recursiva é a última operação a executar, não é preciso guardar um contexto de retorno, então a chamada de função pode ser implementada de maneira muito mais eficiente, e sem usar espaço na memória. Em mais detalhe: a chamada é implementada apenas como uma instrução de desvio em vez do protocolo completo de chamada de função.

Além da eficiência, o fato de a recursividade de cauda usar apenas uma quantidade constante de memória é importante para garantir a corretude das funções recursivas. Caso contrário, uma

chamada recursiva que não entra em *loop* infinito, mas chama a si mesma um grande número de vezes, pode estourar a pilha.

A recursividade de cauda não é importante apenas para funções que entram em *loop* infinito. Vejamos novamente a definição da função `fatorial`:

```
# let rec fatorial n =
    match n with
      0 -> 1
    | 1 -> 1
    | _ -> n * fatorial (n - 1);;
val fatorial : int -> int = <fun>
```

Esta função não apresenta recursividade de cauda, o que pode ser um problema dependendo da situação. Uma técnica para transformá-la de forma que ela apresente recursividade de cauda é o uso de *acumuladores*. Um acumulador é um parâmetro adicional para a função que serve para guardar o resultado intermediário da computação. A função fatorial com um acumulador fica da seguinte forma:

```
# let rec fatorial_ac n ac =
    match n with
      0 | 1 -> ac
    | _ -> fatorial_ac (n-1) (n * ac);;
val fatorial_ac : int -> int -> int = <fun>

# fatorial_ac 5 1;;
- : int = 120
```

É preciso chamar a função passando o valor inicial do acumulador. Neste caso, como a operação é de multiplicação, o acumulador deve começar igual a 1. A avaliação da chamada `fatorial_ac 3 1` demonstra que não é preciso guardar uma quantidade crescente de contexto para obter o resultado:

```
fatorial_ac 3 1
  => fatorial_ac 2 (3 * 1)
  => fatorial_ac 1 (2 * 3 * 1)
  => 6
```

A desvantagem dessa definição é a necessidade de o usuário passar o valor inicial do acumulador, e um valor errado vai afetar o resultado. A função `fatorial_ac` é um detalhe de implementação e não deve ser chamada diretamente. A solução é fazer com que ela seja uma função local dentro da função `fatorial` completa:

```
# let fatorial n =
    let rec fatorial_ac n ac =
      match n with
        0 | 1 -> ac
      | _ -> fatorial_ac (n-1) (n * ac)
    in fatorial_ac n 1;;
val fatorial : int -> int = <fun>
```

A função `fatorial` apenas chama a função interna `fatorial_ac`, passando o valor inicial correto para o acumulador. É interessante notar que agora `fatorial` não é mais recursiva, apenas a função interna.

Em OCaml, as vantagens da recursividade de cauda também se aplicam quando a função não é recursiva: se a função `f` chama a `g` como sua última operação, a chamada a `g` será eficiente e usará uma quantidade constante de espaço. Essa é uma otimização chamada *Tail Call Optimization*, ou TCO.

Como outras otimizações, é importante só se preocupar em transformar uma função para usar a recursividade de cauda se isso for realmente necessário. Para funções que vão se chamar recursivamente poucas vezes, o impacto no desempenho geral do programa é mínimo.

Também existe a questão da corretude, como mencionado antes, mas é mais interessante primeiro verificar que ela gerará um grande número de chamadas recursivas antes de pensar em mudar seu código para usar acumuladores. Mesmo assim, saber fazer essa transformação é importante, já que, na prática, a necessidade de escrever funções recursivas eficientes aparece com frequência em

programas não triviais.

5.4 FUNÇÕES DE PRIMEIRA CLASSE

Em OCaml, assim como na maioria das linguagens funcionais, as funções são tratadas como cidadãos de primeira classe pela linguagem. Isso significa que podemos usar uma função como um valor da linguagem, assim como inteiros ou valores booleanos. Podemos fazer com uma função todas as coisas que fazemos com valores inteiros ou de outros tipos, por exemplo:

- atribuir funções a variáveis;
- passar funções como parâmetros para outras funções;
- retornar funções como resultado de outras funções;
- armazenar funções como componentes de estruturas de dados maiores.

Muitas linguagens incluem alguma forma de trabalhar com funções como valores (por exemplo, os ponteiros para funções em C). Porém, nas linguagens funcionais, o tratamento de funções como valores tende a ser mais transparente e prático.

Esse tipo de manipulação com funções é tão central em OCaml que mesmo coisas como funções de mais de um parâmetro são, na verdade, funções que retornam funções. Funções que recebem funções como parâmetro e/ou retornam outras funções são muitas vezes chamadas de *funções de alta ordem* (em contraste com as funções de *primeira ordem* que não podem receber outras como parâmetro ou retornar funções).

Se as funções são cidadãos de primeira classe na linguagem, deve ser possível especificar um valor funcional (uma expressão que denota uma função) sem dar um nome a ela, assim como o número `5` é um inteiro independente do fato de existir uma variável com

valor `5` ou não.

Funções anônimas (lambda)

Uma *função anônima* é uma expressão funcional que especifica a relação entre a entrada e saída, sem dar um nome a essa relação. Por motivos históricos, as funções anônimas são às vezes chamadas de *lambda*. Em resumo, os motivos históricos são a linguagem LISP original que usou `lambda` como palavra-chave para funções anônimas, além da forte influência do lambda-cálculo na programação funcional.

A sintaxe para uma função anônima com um parâmetro é:

```
fun v -> e
```

O `v` é o nome do parâmetro e `e` é uma expressão que corresponde ao corpo da função. Para um dado argumento de entrada, o resultado da função será o resultado da expressão `e`.

Uma função anônima pode ser entendida como um *literal de função*, ou seja, um valor que denota uma função independente do nome que é dado a ela, assim como o literal inteiro `5` denota um número independente do nome de qualquer variável que receba o valor 5. Isso segue o tema de funções como cidadãos de primeira classe na linguagem, e significa que podemos fazer com uma função anônima qualquer coisa que fazemos com outros valores da linguagem que são funções.

A maior utilidade de uma função é aplicá-la e obter o seu resultado. Seguindo a sintaxe de aplicação de funções em OCaml, basta colocar o argumento após a função, da mesma forma como seria se a função tivesse um nome:

```
# (fun x -> x * x) 5;;
- : int = 25
```

Podemos também atribuir uma função a uma variável:

```
# let q = fun x -> x * x;;
val q : int -> int = <fun>
```

Assim, `q` é uma função que eleva seu argumento ao quadrado:

```
# q 7;;
- : int = 49
```

Na verdade, a sintaxe que usamos até agora para definir funções é um pouco de *açúcar sintático* em cima da ideia de definir variáveis cujo valor é uma função. Ou seja, a função `q` (de *quadrado*) também poderia ser definida como:

```
# let q x = x * x;;
val q : int -> int = <fun>
```

Isso indica que `let f v = e` é apenas uma abreviação para `let f = fun v -> e`. Também podemos definir funções anônimas com mais de um parâmetro, mas nessa altura vale a pena entender como funcionam as funções com mais de um parâmetro em OCaml.

Currificação

Em OCaml, todas as funções têm apenas um parâmetro. Isso pode parecer contraditório considerando todos os exemplos de funções com vários parâmetros que vimos até agora, mas esse é mais um uso do açúcar sintático na linguagem. Funções que parecem ter vários parâmetros, na verdade, são combinações de funções que aceitam, cada uma, apenas um parâmetro.

Vimos que uma função em OCaml é tratada como um valor de primeira classe, o que significa que se pode fazer com uma função tudo que pode ser feito com outros valores. Isso inclui retornar uma função como resultado de outra.

Usando a sintaxe para funções anônimas, vamos definir uma

função `cria_multiplicador`. Ela recebe um número `x` como parâmetro e retorna uma função que multiplica qualquer número por `x`:

```
# let cria_multiplicador x = fun y -> x * y;;
val cria_multiplicador : int -> int -> int = <fun>
```

`cria_multiplicador` retorna uma função que recebe um parâmetro e o multiplica por um número fixo `x`. Com ela, podemos criar novas funções de multiplicação por números específicos:

```
# let mult5 = cria_multiplicador 5;;
val mult5 : int -> int = <fun>

# mult5 7;;
- : int = 35
```

Uma função que retorna funções pode ser útil em muitas situações. Mas atente para o tipo de `cria_multiplicador`, e compare com o tipo da função de multiplicação associada ao operador `*`:

```
# cria_multiplicador;;
- : int -> int -> int = <fun>

# ( * );;
- : int -> int -> int = <fun>
```

O tipo de ambas as funções é exatamente o mesmo: `int -> int -> int`. Como a linguagem é tipada, dois valores de mesmo tipo devem poder ser usados nos mesmos contextos:

```
# cria_multiplicador 5 11;;
- : int = 55
```

Ou seja, `cria_multiplicador` é apenas uma função de multiplicação em OCaml, apesar de ser definida com apenas um parâmetro. Isso é o resultado de uma transformação chamada de *currificação* (do inglês *currying*, em homenagem ao lógico e matemático Haskell Curry). A currificação é uma forma de

representar funções com vários parâmetros usando apenas funções com um parâmetro, e se originou em sistemas lógicos como o lambda-cálculo, em que é mais prático definir apenas funções de um parâmetro.

A ideia da currificação é que uma função com dois parâmetros é na verdade uma função de um parâmetro que retorna outra função que recebe um parâmetro e retorna o resultado final da função. Considere uma função como `mult` que multiplica dois números:

```
# let mult x y = x * y;;
val mult : int -> int -> int = <fun>
```

Já vimos que uma função de um parâmetro é transformada na atribuição de uma variável a uma função anônima de um parâmetro, ou seja, `let f v = e` é transformada em `let f = fun v -> e`. Pensando em `(mult x)` como uma função de um parâmetro, vemos que a transformação de `(mult x) y` é:

```
# let mult x = fun y -> x * y;;
val mult : int -> int -> int = <fun>
```

Já que nesse caso a função é `(mult x)`, o parâmetro é `y` e a expressão é `x * y`. Note que `mult` nessa forma é idêntica à função `cria_multiplicador` que foi vista anteriormente. Agora `mult` é uma função de um parâmetro, e podemos usar a mesma transformação novamente, reescrevendo como:

```
# let mult = fun x -> fun y -> x * y;;
val mult : int -> int -> int = <fun>
```

Essa é a forma "verdadeira" da função `mult`, mesmo quando definida da primeira forma. Nessa última, fica claro que `mult` é uma combinação de duas funções que recebem um parâmetro. Todas as versões mostradas têm o mesmo tipo e funcionam da mesma maneira:

```
# mult 6 7;;
- : int = 42
```

Podemos pensar em uma aplicação, como `mult 6 7`, como sendo `(mult 6) 7`: primeiro ocorre a aplicação de `mult` ao argumento `6`, resultando em uma função de um parâmetro, `(mult 6)`. Ela é aplicada ao argumento `7`, resultando no valor final da chamada.

O tipo de `mult` também faz mais sentido dessa forma: `int -> int -> int` significa `int -> (int -> int)`, ou seja, uma função que recebe um `int` e retorna um valor de tipo `int -> int`. Um valor com este tipo é uma função que recebe um `int` como parâmetro e resulta em um `int`. Obviamente, as mesmas transformações valem para funções com três ou mais parâmetros.

A linguagem OCaml faz essas transformações sintáticas tanto na definição das funções quanto na sua aplicação, tornando prático e corriqueiro o uso da currificação. Também podemos definir funções de múltiplos parâmetros em OCaml usando tuplas, o que ainda é um caso de funções de um parâmetro (mas o argumento é uma tupla):

```
# let mul (x, y) = x * y;;
val mul : int * int -> int = <fun>
```

Mas a vantagem de ter funções currificadas é poder fazer *aplicações parciais*, ou seja, fornecer apenas alguns argumentos a uma função para construir uma nova função. Em OCaml, isso pode ser e é usado com frequência. Por exemplo, podemos criar facilmente uma função que multiplica um número por 5:

```
# let mult5 = ( * ) 5;;
val mult5 : int -> int = <fun>
```

Isso é especialmente útil quando usado conjuntamente com funções que recebem outras como parâmetros. Um exemplo seria multiplicar todos os números de uma lista por 2, usando `List.map`, que veremos na seção *Padrões de recursividade*:

```
# List.map (( * ) 2) [1; 2; 3; 4; 5];;
- : int list = [2; 4; 6; 8; 10]
```

A transformação sintática que faz a currificação com funções de vários parâmetros também se aplica às funções anônimas. É possível definir uma função anônima com dois ou mais parâmetros apenas separando o nome dos parâmetros com espaços, e o compilador OCaml se encarrega de transformar isso em uma sequência de funções anônimas de um parâmetro:

```
# (fun x y -> x * y) 7 6;;
- : int = 42
```

Funções anônimas com function

Existe uma outra forma de definir uma função anônima em OCaml, usando a palavra-chave `function` em vez de `fun`:

```
# function x -> x * x;;
- : int -> int = <fun>
```

Mas há duas diferenças importantes: `function` só pode declarar funções de um parâmetro (ou seja, o compilador não currifica automaticamente):

```
# function x y -> x * y;;
Error: Syntax error
```

A outra diferença é que `function` pode fazer *pattern matching* diretamente no seu único parâmetro, sem usar `match`:

```
# function [] -> true | _ -> false;;
- : 'a list -> bool = <fun>
```

Se usarmos `function` juntamente com um `let` para definir uma nova função, economizamos um pouco de código em relação a usar `match`:

```
# let lista_vazia = function
  | [] -> true
  | _ -> false;;
```

```
val lista_vazia : 'a list -> bool = <fun>
```

O ganho por usar `function` em vez de `match` é pequeno, mas não é raro encontrar usos dessa forma de definir funções em código OCaml. Também é comum para funções com vários parâmetros em que o `match` é feito no último parâmetro. Nesse caso, os anteriores ao último são declarados normalmente no `let`, e o último parâmetro fica a cargo do `function`.

A função a seguir verifica se um valor do tipo `option` contém um valor e este é igual a `x`:

```
# let igual_opcao x = function
  | None -> false
  | Some y -> x = y;;
val igual_opcao : 'a -> 'a option -> bool = <fun>
```

Note que `igual_opcao` recebe dois parâmetros: um valor `x` qualquer e um valor de tipo `option`.

```
# igual_opcao 3 (Some 3);;
- : bool = true
```

O último parâmetro fica implícito no `function`. Podemos considerar que isso atrapalha a leitura da função em um primeiro momento e é uma boa ideia evitar esse tipo de código. Ainda assim, existem alguns usuários da linguagem que escrevem código nesse estilo.

A possibilidade de definir funções anônimas pode parecer pouco interessante à primeira vista, mas a utilidade de criar funções sem precisar dar um nome a elas se torna mais óbvia quando usamos funções de alta ordem que capturam certos padrões comuns de recursividade, como veremos a seguir.

5.5 PADRÕES DE RECURSIVIDADE

Entendendo como escrever funções recursivas baseadas na

estrutura do tipo, como já vimos, fica fácil escrever muitas funções de interesse com estruturas como listas. Na verdade, fica até trivial, e muitas funções escritas terminam com uma estrutura muito similar.

Os padrões que se repetem em funções recursivas podem ser capturados em funções de alta ordem, como normalmente é feito em linguagens funcionais. Embora todo tipo de função possa ser escrito de maneira recursiva, programadores com mais experiência no paradigma funcional tendem a reconhecer quando uma função vai ter uma estrutura comum e usam as funções predefinidas pela linguagem.

Nesta seção, vamos estudar os padrões de recursividade mais importantes e como usar as funções de alta ordem associados a cada um deles.

Mapeamento

Uma situação comum, da qual já vimos alguns exemplos, é transformar todos os elementos em uma coleção da mesma forma, criando uma nova coleção com os resultados. Por exemplo, uma função para elevar todos os números de uma lista ao quadrado pode ser escrita de forma recursiva:

```
# let rec quadrado_lista l =
    match l with
      [] -> []
    | n :: rl -> (n * n) :: quadrado_lista rl;;
val quadrado_lista : int list -> int list = <fun>
```

Uma função para mudar o primeiro caractere de cada string em uma lista para letra maiúscula seria escrita da seguinte forma:

```
# let rec maiusculas_lista ls =
    match ls with
      [] -> []
    | s :: rls -> String.capitalize s :: maiusculas_lista rls;;
val maiusculas_lista : string list -> string list = <fun>
```

As duas funções têm a mesma estrutura. Se a lista de entrada é vazia, a resposta é uma lista vazia. Se a lista é composta pelo *cons* de um item e uma outra lista, aplique uma função ao item da frente e coloque o resultado na frente de uma lista cujo resto é uma chamada recursiva com o resto da lista de entrada como argumento.

O resultado final é transformar uma lista `[n1, n2, ..., nm]` em uma lista `[f n1, f n2, ..., f nm]`, na qual cada elemento é transformado pela mesma função `f`. Vamos escrever uma função que captura esse padrão, recebendo como parâmetros a função de transformação e a lista a ser transformada. Como essa transformação da lista também pode ser pensada como um *mapeamento*, vamos chamá-la de `map`:

```
# let rec map f l =
    match l with
      [] -> []
    | x :: rl -> f x :: map f rl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Essa função é polimórfica, e tem de ser. A função de transformação `f` pode transformar um valor de qualquer tipo em um valor de qualquer outro tipo. Dessa forma, podemos entender o tipo da função `map`: dada uma função `f` que transforma um valor de tipo `'a` em um valor de tipo `'b` e uma lista de valores de tipo `'a`, o resultado é uma lista de valores de tipo `'b`.

Com a função `map`, podemos criar `quadrado_lista` e `maiusculas_lista` de forma muito mais fácil:

```
# let quadrado_lista l = map (fun x -> x * x) l;;
val quadrado_lista : int list -> int list = <fun>

# let maiusculas_lista l = map String.capitalize l;;
val maiusculas_lista : string list -> string list = <fun>
```

Na verdade, essas definições são tão curtas que muitas vezes não compensa criar uma nova função só para isso. Se quisermos elevar todos os elementos de uma lista ao quadrado, podemos usar `map`

diretamente:

```
# map (fun x -> x * x) [1; 2; 3; 4; 5];;
- : int list = [1; 4; 9; 16; 25]
```

Em ambos os casos, os tipos das listas de entrada e saída são os mesmos, mas isso não é necessário. Por exemplo, a seguinte aplicação de `map` calcula o tamanho de cada string em uma lista:

```
# map String.length ["Alys"; "Myau"; "Tyrone"; "Noah (Lutz)"];;
- : int list = [4; 4; 6; 11]
```

Não é preciso escrever a função `map`, pois ela já está definida na biblioteca padrão, no módulo `List`, na maioria das vezes chamada pelo seu nome completo, `List.map`, como já visto em alguns exemplos anteriores. A ideia de mapeamento pode ser e é aplicada a outros tipos de coleções, embora as listas sejam o exemplo mais típico.

Filtragem

Algumas vezes queremos selecionar alguns elementos de uma coleção, de acordo com algum teste (ou predicado). Por exemplo, selecionar apenas os elementos pares de uma lista de números:

```
# let par n = (n mod 2) = 0;;
val par : int -> bool = <fun>

# let rec pares_lista l =
    match l with
      [] -> []
    | n :: rl when par n -> n :: pares_lista rl
    | n :: rl -> pares_lista rl;;
val pares_lista : int list -> int list = <fun>
```

Primeiro definimos uma função para detectar se um número é par. Depois a função `pares_lista` é definida de forma que, se o elemento na frente da lista é par (testado com uma guarda), ele é incluído no resultado; caso contrário, o elemento não é incluído (terceiro caso). A função `pares_lista` funciona como esperado:

```
# pares_lista [3; 11; 26; 13; 2; 0];;
- : int list = [26; 2; 0]
```

Podemos abstrair a estrutura da função `pares_lista`, generalizando o teste a ser aplicado. Como a ideia é a de *filtrar* uma lista de acordo com algum critério, a função será chamada de `filtrar`:

```
# let rec filtrar p l =
    match l with
      [] -> []
    | x :: rl when p x -> x :: filtrar p rl
    | x :: rl -> filtrar p rl;;
val filtrar : ('a -> bool) -> 'a list -> 'a list = <fun>
```

O tipo da função `filtrar` é polimórfico com apenas um tipo variável: dada uma função de teste para um tipo `'a` (retornando um booleano) e uma lista de valores de tipo `'a`, retorna uma lista de tipo `'a` apenas com os elementos da lista de entrada para os quais o teste `p` retorna `true`. Com isso, fica trivial filtrar elementos pares:

```
# filtrar par [3; 11; 26; 13; 2; 0];;
- : int list = [26; 2; 0]
```

Mais uma vez, essa função já está definida na biblioteca padrão da linguagem como a função `filter` no módulo `List` (normalmente chamada com nome `List.filter`). A ideia de filtragem também pode ser aplicada em outras coleções.

Redução (fold)

Outra situação comum é querer calcular um valor a partir de todos os elementos de uma lista, por exemplo, a soma de todos os elementos. Já vimos o exemplo da função `soma_lista`:

```
# let rec soma_lista l =
    match l with
      [] -> 0
    | x :: rl -> x + soma_lista rl;;
val soma_lista : int list -> int = <fun>
```

É preciso estabelecer um valor inicial, no caso zero, e a forma de combinar os valores. A função de combinação recebe dois parâmetros de entrada, combinando o valor acumulado até o momento com o próximo elemento da lista. Generalizando a função `soma_lista` para outros casos, escrevemos a função `reducao`:

```
# let rec reducao f i l =
    match l with
      [] -> i
    | x :: rl -> f x (reducao f i rl);;
val reducao : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

O nome da função vem da nomenclatura da programação científica (é a ideia de *reduzir* a informação em um *array* a um só número). O tipo de `reducao` indica como ela funciona: dada uma função que combina dois parâmetros (um valor de tipo `'a` e um valor de tipo `'b`, resultando em um valor de tipo `'b`), um valor inicial do tipo `'b`, e uma lista de elementos de tipo `'a`, combine o valor inicial com todos os elementos da lista, um de cada vez, e obtenha o valor final da redução. Mais um exemplo em que o tipo ajuda a entender o que a função faz.

Com essa função, é simples calcular a soma de todos os elementos de uma lista:

```
reducao (+) 0 [1; 2; 3; 4];;
- : int = 10
```

A função de combinação, nesse caso, é a soma de inteiros `(+)`.

A redução de uma lista a um resultado é chamada na programação funcional de *fold* (palavra em inglês para *dobra*, ou *dobrar*), que reflete a ideia de dobrar a lista sobre ela mesma para chegar a um resultado compacto. Mas a forma como foi feito o *fold* na função `reducao` não é a única possível. Para ver isso, vamos visualizar a avaliação da expressão `reducao (+) [1; 2; 3]`, lembrando de que `(+) x y` é o mesmo que `x + y` na sintaxe da linguagem OCaml:

```
reducao (+) 0 [1; 2; 3]
  => (+) 1 (reducao (+) 0 [2; 3]) => 1 + (reducao (+) 0 [2; 3])
  => 1 + (2 + (reducao (+) 0 [3]))
  => 1 + (2 + (3 + (reducao (+) 0 [])))
  => 1 + (2 + (3 + 0))
```

A partir daí, fica fácil entender como o resultado é calculado. Mas uma característica interessante da avaliação é a ordem das operações. O valor inicial da redução, que é o zero, é somado com o último elemento da lista (`3`), depois o resultado é somado ao penúltimo elemento da lista (`2`) e, por fim, somado ao primeiro elemento da lista.

Pela estrutura de parênteses, dá para ver que a operação nessa redução é aplicada aos elementos da direita para a esquerda. Uma outra possibilidade para `reducao` seria aplicar a operação da esquerda para a direita:

```
# let rec reducao_esq f i l =
    match l with
      [] -> i
    | x :: rl -> reducao_esq f (f i x) rl;;
val reducao_esq : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

No caso da soma, agrupar as operações da esquerda para a direita ou da direita para a esquerda não faz diferença, já que a adição é associativa. De qualquer forma, a avaliação de uma expressão usando `reducao_esq` prossegue da seguinte maneira:

```
reducao_esq (+) 0 [1; 2; 3]
  => reducao_esq (+) (0 + 1) [2; 3]
  => reducao_esq (+) ((0 + 1) + 2) [3]
  => reducao_esq (+) (((0 + 1) + 2) + 3) []
  => (((0 + 1) + 2) + 3)
```

Comparando esse exemplo com a avaliação do exemplo usando `reducao` , fica clara a diferença na associação das operações. Podemos dizer que a função `reducao` é uma redução *à direita*, e `reducao_esq` é uma redução *à esquerda*. As funções similares definidas na biblioteca padrão são chamadas de `List.fold_right`

(redução à direita) e `List.fold_left` (redução à esquerda), respectivamente.

5.6 TIPOS COMO FONTE DE INFORMAÇÃO

Como discutimos no capítulo *Registros e variantes*, a estrutura recursiva de um tipo ajuda a determinar a estrutura dos algoritmos que processam os valores dele. Do outro lado, o tipo de uma função nos dá informações sobre o comportamento da função, principalmente quando é polimórfico. Vimos alguns exemplos disso nesse capítulo com as funções de alta ordem, como `map` e `fold`.

Se sabemos que uma função `f` tem tipo `int -> int`, mas não conhecemos o seu código, podemos deduzir muito pouco sobre `f`. Mesmo assim, temos garantia de que, se `f` compila, ela retorna um valor inteiro sempre que recebe um inteiro como parâmetro. A verificação de tipos do compilador garante isso não só para o código da própria função `f`, como também garante que todo código que chama `f` passa um inteiro como parâmetro.

Podemos pensar no verificador de tipos como um programa que prova teoremas como "a função `f` sempre retorna um inteiro". Entretanto, não podemos deduzir nada sobre o código de `f`, que pode utilizar qualquer combinação de constantes e operações inteiras para calcular seu resultado. O `f` pode ignorar o parâmetro de entrada e retornar um valor constante, ou pode fazer uma operação matemática complexa para calcular a saída a partir da entrada.

Quando uma função tem tipo polimórfico, normalmente podemos deduzir mais informações sobre ela, justamente por causa da generalidade dos tipos envolvidos. Um exemplo simples é uma função `g` de tipo `'a -> 'a`.

O `g` deve receber um parâmetro que é um valor de qualquer tipo `'a`, e retornar um valor de mesmo tipo. Como o tipo de entrada pode ser qualquer um, incluindo tipos que serão criados no futuro, `g` não pode usar nenhuma operação sobre o parâmetro de entrada, já que não existe nenhuma operação ou função que funcione em valores de todos os tipos.

Mesmo ignorar o parâmetro de entrada e retornar um valor constante do tipo `'a` é impossível, pois `g` não sabe como especificar um valor constante que sirva para todos os tipos ao mesmo tempo. Isso significa que a única coisa que `g` pode fazer é retornar o próprio parâmetro. Podemos escrever essa função facilmente:

```
# let g x = x;;
val g : 'a -> 'a = <fun>
```

De fato, `g` tem tipo `'a -> 'a`. Toda função que tenha esse tipo será igual a `g`. Essa é a única possibilidade. Nesse caso, só ao ver o tipo podemos deduzir até mesmo o seu código completo.

Algo parecido acontece com `List.map`:

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

A função de mapeamento recebe uma função de transformação, de tipo `'a -> 'b` e uma lista de valores `'a`, retornando uma lista de valores `'b`. A única forma que `List.map` tem de obter valores de tipo `'a` é selecionando algum elemento da lista de entrada, e a única forma de produzir um valor de tipo `'b` é chamando a função de transformação, passada como parâmetro.

Isso não determina completamente o que `List.map` faz, mas dá uma boa indicação de que a lista retornada pela função é obtida pela transformação de cada elemento da lista de entrada, usando a função de transformação passada. Da mesma forma, podemos

analisar o tipo de outras funções polimórficas, com listas ou outras estruturas de dados.

Essa é uma vantagem de se trabalhar com tipos estáticos, e representa apenas as informações genéricas que podem ser obtidas do tipo de uma função. O programador também pode ajudar no processo de obter informações a partir dos tipos, ao projetar tipos para seu programa com nome e estrutura informativos.

Teoremas de graça

A ideia de obter informações sobre uma função a partir de seu tipo é explicada em mais detalhes no artigo *Theorems for free!* (teoremas de graça), escrito por Philip Wadler. Os teoremas gratuitos são exatamente aqueles que podem ser determinados somente pelo tipo da função, sem precisar analisar seu código.

Um detalhe importante é que eles podem ser determinados automaticamente, de maneira simples (o artigo detalha o algoritmo para isso). Claro que isso só funciona se temos garantia de que o compilador faz uma verificação de tipos completa e correta.

Apesar de a palavra *teorema* dar a impressão de que isso é importante apenas no meio acadêmico, vimos que informalmente isso significa que podemos deduzir fatos sobre funções a partir de seu tipo que podem até nos ajudar a entender o que a função faz. Os teoremas determinadas para funções também podem ser usados em várias aplicações, por exemplo em otimizações no compilador ou para fazer verificação de programas (determinar sua corretude formalmente).

Uma aplicação interessante dos teoremas gratuitos é derivar automaticamente, a partir do tipo de uma função, testes de unidade que ela deve passar, ou gerar casos específicos para serem testados. Uma ideia similar é usada nas bibliotecas de teste similares a QuickCheck.

5.7 DOIS OPERADORES PARA APLICAR

FUNÇÕES

Em uma linguagem funcional, com funções de primeira classe, a aplicação de funções se torna uma das operações mais importantes. Na programação funcional pura, praticamente tudo é feito usando apenas a aplicação de funções a argumentos que podem ser constantes ou outras funções.

A importância da aplicação (ou chamada) de funções faz com que seja útil ter certas notações que não são estritamente necessárias, mas ajudam em muitas situações. Existem dois operadores em OCaml que entram nessa categoria: `@@` e `|>`.

O `@@` é o operador de aplicação. Sua definição (já existente na biblioteca padrão) é:

```
# let ( @@ ) f x = f x;;
val ( @@ ) : ('a -> 'b) -> 'a -> 'b = <fun>
```

Ou seja, dada uma função `f` e um argumento `x`, aplique `f` a `x`. O operador parece redundante, já que `f @@ x` é o mesmo que `f x`. Entretanto, ele existe por uma questão de conveniência sintática: muitas vezes queremos aplicar uma função ao resultado de outra (na matemática, isso é a *composição* de funções).

Se `f` e `g` são funções, `f g x` não significa `f (g x)`, mas sim uma chamada a `f` com dois argumentos, `g` e `x`. Podemos usar parênteses para separar `g x`, mas às vezes a aplicação `g x` é longa e usar parênteses reduz a legibilidade.

O operador `@@` tem baixa precedência, de forma que `f @@ g x` significa `f (g x)`, como queremos, sem precisar usar parênteses. É uma conveniência cuja utilidade fica mais clara à medida que se programa mais no estilo funcional (e que vamos ver em vários exemplos no resto do livro). Por exemplo, digamos que precisamos implementar uma função que retorna o quadrado da

soma de uma lista de números. A função `quadrado` já foi usada em outros exemplos antes:

```
let quadrado x = x * x
```

Vejamos como fica a função `quad_soma` sem o uso do `@@`:

```
let quad_soma l = quadrado (List.fold_left (+) 0 l)
```

E agora com o `@@`:

```
let quad_soma l = quadrado @@ List.fold_left (+) 0 l
```

Nesse caso, não é uma diferença muito grande, mas não precisamos colocar parênteses ao redor do `fold`, o que melhora a leitura. Usar `@@` se torna ainda mais vantajoso se tivermos uma cadeia de funções. Escrever uma cadeia na forma `f @@ g @@ h x` é melhor que escrever `f (g (h x))`, e a diferença fica maior para cadeias maiores.

Cadeias de funções, em que o resultado de uma é usada como entrada de outra, são comuns na programação funcional. Em alguns contextos, essas cadeias são chamadas de *pipelines*, como na computação gráfica. O problema de uma cadeia como `f @@ g @@ h x` é que lemos da esquerda para a direita, porém o fluxo dos dados na expressão é da direita para a esquerda: `x` primeiro é passada para `h`, o resultado disso é passado para `g`, e assim por diante.

Nesses casos, pode ser mais legível escrever a cadeia de funções no mesmo sentido do fluxo das informações. Para isso, existe o operador `|>`. Ele pode ser definido como:

```
# let ( |> ) x f = f x;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

Ou seja, dado um argumento `x` e uma função `f`, aplique `f` a `x`. Não é preciso definir esse operador, pois ele já faz parte da biblioteca padrão. Usando `|>`, podemos escrever a cadeia `f @@ g`

`@@ h x` na forma de um *pipeline* como `x |> h |> g |> f`, que segue o sentido do fluxo de dados e é mais legível. O operador indica o sentido do fluxo. Veremos exemplos de uso desse operador adiante no livro.

5.8 FUNÇÕES DE ALTA ORDEM EM ÁRVORES

Podemos definir funções de alta ordem como `map` e `filter` para outras estruturas além de listas. Nesta seção, usaremos como exemplo o tipo de árvores polimórficas visto na seção *Árvores polimórficas e valores opcionais*, cuja definição repetimos aqui:

```
type 'a arvore = Folha | No of 'a arvore * 'a * 'a arvore
```

O mapeamento de uma árvore deverá construir uma árvore de mesma estrutura, mas com cada elemento `x` trocado por `f x`, em que `f` é a função que queremos mapear sobre a árvore. A função `map_arvore` segue a estrutura do tipo:

```
# let rec map_arvore f a =
    match a with
    | Folha -> Folha
    | No (ae, x, ad) ->
        No (map_arvore f ae, f x, map_arvore f ad);;
val map_arvore : ('a -> 'b) -> 'a arvore -> 'b arvore = <fun>
```

O tipo da função `map_arvore` mostra o que ela faz: dada uma função que transforma valores do tipo `'a` em valores do tipo `'b`, `map_arvore` transforma uma árvore de valores de tipo `'a` em uma árvore de valores do tipo `'b` (essa é uma descrição currificada do tipo da função). Com isso, podemos facilmente duplicar o valor de todos os valores em uma árvore de inteiros (usando a árvore `a2` da seção *Árvores polimórficas e valores opcionais*):

```
 # let a2 =
   No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha));;
val a2 : int arvore =
   No (No (Folha, 17, Folha), 21, No (Folha, 42, Folha))
```

```
# map_arvore (( * ) 2) a2;;
- : int arvore =
  No (No (Folha, 34, Folha), 42, No (Folha, 84, Folha))
```

Para fazer redução ou `fold` em árvores, temos várias opções para organizar a ordem das operações. A redução em uma lista usa o valor atual da redução e o valor do próximo elemento da lista. O conceito de próximo elemento em uma árvore não é bem definido, embora possamos seguir uma ordem de travessia, como pré-ordem ou pós-ordem. Em vez disso, vamos aqui mudar a estrutura da redução: a função de redução `f` deve receber três parâmetros, um sendo o valor do nó atual e os outros dois sendo o resultado da redução nas duas subárvores abaixo.

O `f` combina esses valores em um resultado que é passado para os antecedentes do nó atual. Como o `fold` recebe um valor inicial, o resultado da redução em uma folha é exatamente esse valor:

```
# let rec fold_arvore f arv ini =
    match arv with
    | Folha -> ini
    | No (ae, x, ad) ->
      f x (fold_arvore f ae ini) (fold_arvore f ad ini);;
val fold_arvore :
  ('a -> 'b -> 'b -> 'b) -> 'a arvore -> 'b -> 'b = <fun>
```

Com essa função, podemos facilmente calcular a soma de todos os valores em uma árvore de inteiros (mais uma vez usando a árvore `a2`). A função de redução deve simplesmente somar os três argumentos que recebe:

```
# fold_arvore (fun x r1 r2 -> x + r1 + r2) a2 0;;
- : int = 80
```

De forma semelhante, poderíamos definir uma função de filtragem em árvores, embora nesse caso também teríamos de pensar o que fazer com elementos que não satisfazem o predicado: poderíamos eliminar toda a subárvore começando com o elemento que não satisfaz o predicado, ou procurar nela por um elemento que

satisfaz o predicado para colocá-lo no lugar, o que iria requerer uma reestruturação de toda a subárvore. A primeira opção é a mais simples, mas deixamos sua escrita como um exercício para o leitor.

Agora que vimos os princípios da programação funcional, é interessante colocá-los em prática em programas um pouco maiores. No próximo capítulo, isso será feito com um tipo de aplicação bastante comum em linguagens funcionais: a criação de interpretadores e compiladores.

Capítulo 6

EXEMPLO: INTERPRETADOR E COMPILADOR

Até agora, vimos exemplos de código curtos, criados para ilustrar alguma característica específica da linguagem OCaml. Neste capítulo, vamos examinar um programa um pouco mais longo, integrando várias características da linguagem.

Compiladores, interpretadores e outras ferramentas que processam código em linguagens de programação são uma classe de aplicação bastante comum para linguagens funcionais. Veremos como exemplos um interpretador e um compilador para uma linguagem simples de expressões aritméticas.

Este capítulo não é necessário para entender os seguintes. Se estiver com pressa para aprender mais sobre a linguagem OCaml, não há problema em pular para o próximo.

6.1 EXPRESSÕES ARITMÉTICAS

As expressões aritméticas como `x * 3 + 2` formam uma parte importante da maioria das linguagens de programação, desde a linguagem FORTRAN original. A principal inovação da linguagem FORTRAN (*FORmula TRANSlator*) foi justamente permitir aos programadores que escrevessem fórmulas usando a sintaxe

tradicional da matemática em vez de sequências de comandos, como `add` e `mult`.

Nas linguagens imperativas, existe uma sublinguagem de expressões que está inserida na linguagem completa, que é uma linguagem de comandos. Como já vimos, OCaml é uma linguagem orientada a expressões, na qual não existe separação entre expressões e comandos: existem apenas expressões. Desta forma, começando apenas com uma pequena linguagem de expressões, podemos expandi-la para uma linguagem imperativa (adicionando comandos), ou para uma linguagem funcional (adicionando novas formas de expressão).

Aqui vamos começar apenas com as expressões aritméticas, e usando apenas operandos inteiros. Algumas expressões que se encaixam nessa linguagem são `5 * 3 + 2` e `(9 - 4) + 6`. Essa linguagem é excessivamente simples, mas serve para ilustrar as ideias principais de como criar interpretadores e compiladores.

Um pouco sobre compiladores e interpretadores

Um compilador é um tradutor: dado um programa escrito em uma linguagem de programação, o compilador gera um programa equivalente em outra linguagem. Geralmente, a linguagem de entrada do compilador é uma linguagem de alto nível (Java, OCaml), e a linguagem de saída é uma linguagem de baixo nível diretamente executável por alguma plataforma (linguagem de máquina para um processador, *bytecodes* para a Máquina Virtual Java). O propósito é transformar o código de alto nível de um programa em um executável.

Um interpretador lê um programa e executa-o imediatamente. Para diferenciar de como funciona um compilador, podemos definir que um interpretador *puro* realiza todas as tarefas de análise do programa de entrada toda vez que ele precisa ser executado.

Na prática, isso seria ineficiente, portanto, a maioria dos interpretadores traduz o programa de entrada para um formato interno mais fácil de executar (ou seja, de certa forma todo interpretador também faz compilação). Uma visão de alto nível da diferença entre compiladores e interpretadores pode ser vista na figura:

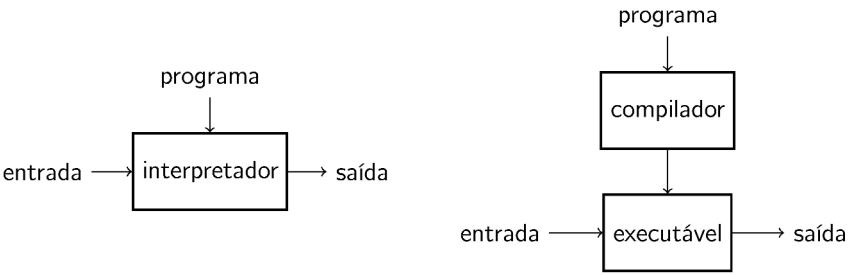

Figura 6.1: Funcionamento de um interpretador e um compilador

Um compilador pode ser estruturado em duas grandes etapas: na etapa de *análise*, o programa de entrada é analisado para que seu significado seja estabelecido; na etapa de *síntese*, as informações sobre o significado do programa são usadas para gerar o código de destino na linguagem de saída. Essa estrutura é mostrada na figura a seguir.

Dessa forma, o compilador funciona de maneira análoga a um tradutor humano que deve traduzir um texto escrito em uma língua para a outra: a etapa de análise é a leitura do texto original, para determinar seu significado e suas nuances, e a etapa de síntese é a escrita do texto traduzido na língua de destino. Em um interpretador puro, a fase de síntese é apenas a execução direta do programa.

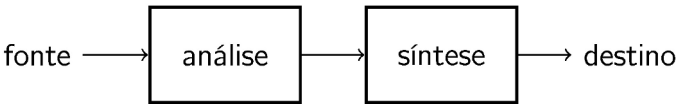

Figura 6.2: Estrutura geral de um compilador

Análise sintática e sintaxe abstrata

O primeiro problema ao escrever um compilador ou interpretador é ler e analisar a entrada. O código-fonte normalmente está contido em um arquivo que é passado como entrada do compilador. Este passa inicialmente por duas fases da etapa de análise: *análise léxica* e *análise sintática*.

O propósito dessas fases é obter a estrutura do programa de entrada, a partir da representação como uma sequência de caracteres. O resultado é normalmente representado como uma árvore, a chamada *árvore de sintaxe abstrata*. A sintaxe abstrata captura apenas a estrutura do programa de entrada, e não os detalhes como parênteses, precedência e pontuação.

A árvore de sintaxe abstrata para a expressão `(4 + 3) * 2 + 5` é mostrada na figura seguinte. Na árvore, não é necessário representar os parênteses, pois sua estrutura já captura a precedência.

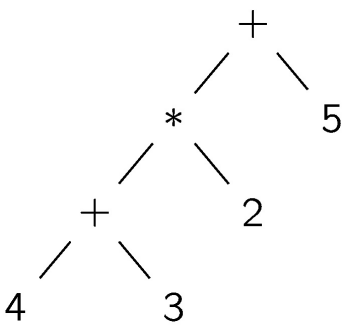

Figura 6.3: Árvore de sintaxe abstrata para a expressão (4 + 3) * 2 + 5

Como vimos no capítulo *Registros e variantes*, os ADTs em OCaml funcionam muito bem como representação de árvores, e esse é um dos motivos por que a linguagem OCaml é uma boa opção para escrever interpretadores e compiladores.

Os exemplos que veremos neste capítulo trabalham apenas com a sintaxe abstrata das expressões, evitando a análise sintática para se concentrar nos mecanismos de execução e tradução. O exemplo completo no repositório inclui um analisador sintático para a linguagem (o código pode ser encontrado no site do livro, cujo endereço está no prefácio).

Definição da sintaxe abstrata

Começamos definindo um tipo variante que captura a sintaxe abstrata da linguagem de expressões. A linguagem inclui números inteiros (do tipo `int`) como operandos e três operações: adição, subtração e multiplicação. O ADT que representa uma expressão é:

```
type exp =
  | Const of int
  | Soma of exp * exp
  | Sub of exp * exp
  | Mult of exp * exp
```

Por exemplo, a expressão `4 + 3` é representada pelo valor `Soma (Const 4, Const 3)`. Alguns outros exemplos de expressão e árvore sintática correspondentes (assumimos que a multiplicação tem precedência sobre a soma):

1. Expressão: `4 + 3 * 2`

   Árvore: `Soma (Const 4, Mult (Const 3, Const 2))`

2. Expressão: `(4 + 3) * 2`

   Árvore: `Mult (Soma (Const 4, Const 3), Const 2)`

3. Expressão: `(4 + 3) * 2 + 5`

   Árvore: `Soma (Mult (Soma (Const 4, Const 3), Const 2), Const 5)`

Como dá para ver pelos exemplos, a sintaxe abstrata se torna difícil de ler para expressões maiores. Por isso, é comum definir uma forma de fazer *pretty printing*, que é mais ou menos o inverso da análise sintática: a partir da sintaxe abstrata, obter uma representação na sintaxe concreta da linguagem. É fácil definir uma função recursiva que faz uma versão simplificada de *pretty printing*:

```
let rec print e =
  match e with
    Const n -> string_of_int n
  | Soma (e1, e2) ->
     Printf.sprintf "(%s + %s)" (print e1) (print e2)
  | Sub (e1, e2) ->
     Printf.sprintf "(%s - %s)" (print e1) (print e2)
  | Mult (e1, e2) ->
     Printf.sprintf "(%s * %s)" (print e1) (print e2)
```

Podemos testar essa função no REPL:

```
# #use "exp.ml";;

# print @@ Mult(Soma(Const 4, Const 3), Const 2);;
- : string = "((4 + 3) * 2)"

# print @@
  Soma(Mult(Soma(Const 4, Const 3), Const 2), Const 5);;
- : string = "(((4 + 3) * 2) + 5)"
```

A função `print` insere parênteses em torno de todas as operações, mesmo quando não é necessário. Seria mais legível inserir parênteses apenas quando a ordem das operações não seguir a precedência e associatividade dos operadores, porém isso tornaria a função `print` um pouco mais complicada (fica como exercício para o leitor).

6.2 INTERPRETAÇÃO

Considerando a sintaxe abstrata apresentada, um interpretador para a linguagem de expressões pode ser escrito facilmente como uma função recursiva usando `match`:

```
let rec eval e =
  match e with
    Const n -> n
  | Adic (e1, e2) -> eval e1 + eval e2
  | Sub (e1, e2) -> eval e1 - eval e2
  | Mult (e1, e2) -> eval e1 * eval e2
```

O interpretador consiste na função de avaliação `eval`, que obtém corretamente o valor de qualquer expressão que segue a sintaxe abstrata do tipo `exp`:

```
 # eval (Soma(Mult(Soma(Const 4, Const 3), Const 2), Const 5));;
- : int = 19
```

A função `eval` demonstra a facilidade de processar expressões em sintaxe abstrata. A representação das expressões usando o tipo `exp` é difícil de ler e de escrever, mas faz com que processar as expressões se torne simples. A questão é usar a representação mais adequada para cada aplicação: *strings* para os usuários que escrevem os programas, mas árvores de sintaxe abstrata para algumas partes do compilador.

Para outras tarefas de compilação, por exemplo na otimização, pode ser mais eficiente usar outras representações. No geral, essas representações do programa que não são nem a linguagem de entrada nem a linguagem de saída são chamadas de *representações intermediárias*.

Já que um interpretador para essa linguagem é bastante simples, vamos passar para um compilador. Como vimos (por exemplo na figura *Funcionamento de um interpretador e um compilador*), um compilador traduz o programa de entrada para alguma linguagem de baixo nível que pode ser executada em alguma plataforma. Para simplificar, vamos trabalhar com uma plataforma fictícia: uma máquina de pilha simples.

6.3 UMA MÁQUINA DE PILHA

Existem dois tipos principais de arquiteturas de máquinas de execução: máquinas de pilha e máquinas de registrador. Podemos dizer que uma máquina de pilha executa instruções que obtêm os dados de entrada de uma *pilha*, e armazenam os dados de saída na mesma pilha. Esta é um dispositivo de memória que está organizada para seguir a estratégia LIFO (*Last In, First Out*, ou o último a entrar é o primeiro a sair) de armazenamento.

Já uma máquina de registradores executa instruções que obtêm dados de entrada dos registradores da máquina e armazenam os dados de saída também em registradores. Os registradores são memórias especializadas que normalmente armazenam uma pequena quantidade de dados. Em arquiteturas de *hardware*, os registradores fazem parte do processador e possibilitam um acesso extremamente eficiente.

A maioria dos processadores (CPUs) atuais define uma arquitetura de nível de instrução (*Instruction Set Architecture*) baseada no modelo de máquina de registradores. Máquinas de pilha são um modelo comum para Máquinas Virtuais implementadas em software, como a Máquina Virtual Java (JVM). O sistema de tempo de execução da plataforma .NET (*Common Language Runtime*) também segue um modelo de máquina de pilha.

Para a linguagem de expressões, vamos definir uma máquina de pilha simples com quatro instruções: uma para empilhar uma constante e três para realizar as operações da linguagem (soma, subtração e multiplicação). A instrução de empilhar insere uma constante na frente da pilha, e as instruções de operação desempilham dois números da pilha e empilham o resultado da operação.

Por exemplo, a instrução de soma desempilha dois números e empilha a soma deles no lugar. A figura a seguir mostra o comportamento da pilha durante a execução de três instruções da

máquina, duas de empilhar e uma soma.

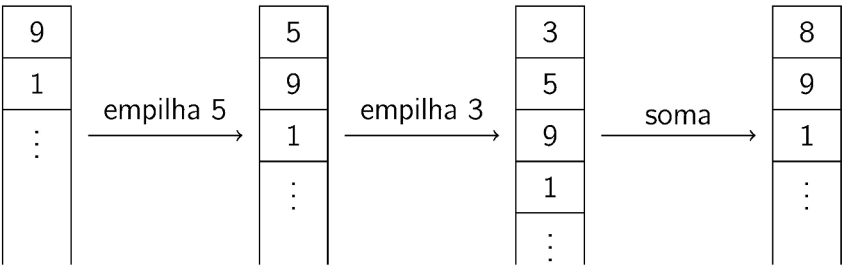

Figura 6.4: Exemplo de execução de um programa na máquina de pilha

Começamos definindo os tipos relativos a essa máquina de pilha. O tipo `operacao` especifica uma das três operações aritméticas da máquina:

```
type operacao = OpSoma | OpSub | OpMult
```

Uma instrução da máquina de pilha é uma instrução para empilhar ou uma operação. No primeiro caso, é preciso especificar qual a constante para empilhar, e no segundo, é preciso especificar qual é a operação:

```
type instrucao =
  | Empilha of int
  | Oper of operacao
```

Por fim, um programa da máquina de pilha é apenas uma lista de instruções, e a pilha da máquina será representada por uma lista de inteiros. Note que as operações usuais de uma lista funcionam perfeitamente para uma pilha: adicionar um elemento na frente da lista é empilhar, e retirar o elemento da frente (usando `List.hd` ou um `match`) corresponde a desempilhar.

```
type programa = instrucao list

type pilha = int list
```

Essa máquina de pilha que está sendo descrita é uma máquina

virtual; ela não existe em forma de hardware para executar essas instruções simples. Como uma máquina virtual, é preciso criar um sistema de execução para programas da máquina. Felizmente, como a máquina tem instruções muito simples, um sistema de execução para ela é fácil de programar.

Primeiro vamos definir duas funções auxiliares para ajudar na execução de operações. Uma operação precisa desempilhar dois valores da pilha para usar como operandos. A função `operandos` faz isso: dada a pilha atual, ela retorna um par com os dois elementos no topo da pilha (se existem) e o resto da pilha. O retorno é um tipo `option` e, se a pilha tem menos de dois elementos, ela retorna `None`:

```
let operandos p =
  match p with
  | [] -> None
  | _ :: [] -> None
  | n1 :: n2 :: r -> Some ((n1, n2), r)
```

A segunda função utilitária simplesmente relaciona cada possível operador da máquina com o operador correspondente em OCaml. Por exemplo, o operador `OpSoma` da máquina de pilha corresponde ao operador `+` em OCaml. Neste caso, estamos aproveitando uma semelhança entre a *linguagem objeto* e a *metalinguagem*.

```
let oper o =
  match o with
  | OpSoma -> (+)
  | OpSub -> (-)
  | OpMult -> ( * )
```

Linguagem objeto e metalinguagem

Quando se fala de compiladores e interpretadores, é comum termos várias linguagens envolvidas. Um compilador (ou interpretador) é escrito em uma linguagem de programação L, e recebe como entrada programas escritos em uma linguagem de programação que geralmente é diferente de L. O compilador produz como saída um programa em uma terceira linguagem, muitas vezes diferente das outras duas.

Para evitar confusão entre as linguagens, usamos uma nomenclatura que vem da lógica matemática: a linguagem que é manipulada pelo compilador (ou interpretador) é chamada de *linguagem objeto*, enquanto que a linguagem na qual o compilador é escrito (ou seja, a linguagem usada para manipular a linguagem objeto) é chamada de *metalinguagem*. No exemplo deste capítulo, a linguagem objeto é a linguagem de expressões apresentada no começo do capítulo, e a metalinguagem é a linguagem OCaml.

Agora podemos escrever a função que executa uma instrução da máquina. Dada a situação atual da pilha da máquina, a execução de uma instrução vai resultar em um novo estado para a pilha. Aqui representamos isso construindo uma nova pilha com o próximo estado da máquina.

Se a instrução for de empilhar, a pilha resultante será igual à anterior com um inteiro inserido na frente. Se for um operador, obtemos os operandos e o operador, e empilhamos o resultado da operação. O único cuidado é o que fazer caso a instrução a executar seja uma operação, mas a pilha não contenha dois operandos; para simplificar, a função de execução simplesmente retorna a pilha

anterior inalterada, mas seria melhor usar uma exceção (como veremos no capítulo *Características imperativas*).

```
let exec_inst p inst =
  match inst with
    Empilha n -> n :: p
  | Oper o ->
     match operandos p with
       None -> p
     | Some ((n1, n2), r) ->
        let op = oper o in
        (op n1 n2) :: r
```

Finalmente, como um programa é apenas uma lista de instruções, executar um é executar uma sequência de instruções, mantendo a pilha de forma que a pilha resultante de uma instrução deve ser a entrada para a próxima instrução. Isso pode ser feito com uma função recursiva, mas basicamente é um `fold` em cima da lista de instruções usando `exec_inst`:

```
let rec exec_prog p =
  List.fold_left exec_inst [] p
```

Podemos agora ver o resultado da execução de um programa na máquina:

```
# exec_prog [Empilha 5; Empilha 3; Oper OpSoma];;
- : int list = [8]
```

O resultado da execução do programa é uma pilha, com o valor resultante no topo. Poderíamos escrever uma função auxiliar que retorna apenas o resultado do programa. O problema é que, se o programa for mal formado (por exemplo, se houver uma instrução para a qual não existem operandos na pilha), não há resultado para retornar. Dessa forma, a função auxiliar deveria retornar um valor opcional, e isso não seria mais prático do que verificar se a pilha retornada por `exec_prog` está vazia.

Agora que temos uma máquina virtual para servir como plataforma de execução, podemos criar um compilador para a

linguagem de expressões, traduzindo-a para a linguagem da máquina de pilha.

6.4 COMPILAÇÃO

A compilação de uma expressão (na forma de uma árvore sintática abstrata) para um programa da máquina de pilha é feita por uma travessia da árvore da expressão, traduzindo cada parte em uma função recursiva.

A expressão constante é traduzida para um programa com uma instrução que empilha o mesmo valor. Para uma operação com duas subexpressões como operandos, é preciso traduzir recursivamente cada subexpressão, concatenando os programas traduzidos, e finalizar com uma instrução para a operação em questão. A função de compilação, portanto, pode ser escrita da seguinte forma:

```
let rec compila e =
  match e with
  | Const n       -> [Empilha n]
  | Soma (e1, e2) -> (compila e2) @ (compila e1) @ [Oper OpSoma]
  | Sub (e1, e2)  -> (compila e2) @ (compila e1) @ [Oper OpSub]
  | Mult (e1, e2) -> (compila e2) @ (compila e1) @ [Oper OpMult]
```

Veja que os operandos são colocados na lista de instruções na ordem inversa. Isso acontece por causa do uso da pilha.

Por exemplo, se a expressão for `7 - 2`, o programa traduzido precisa empilhar primeiro o 2, depois o 7, e em seguida fazer a subtração. A ordem é essa porque, na máquina de pilha, o operando esquerdo é o que está no topo dela (como pode ser visto na função `exec_inst`). Por isso, a ordem é invertida na sequência de instruções. Podemos verificar a compilação de algumas expressões:

```
# compila @@ Soma (Const 5, Const 3);;
- : instrucao list = [Empilha 3; Empilha 5; Oper OpSoma]

# let e2 =
```

```
    Soma(Mult(Soma(Const 4, Const 3), Const 2), Const 5);;
val e2 : exp =
  Soma (Mult (Soma (Const 4, Const 3), Const 2), Const 5)

# compila e2;;
- : instrucao list =
[Empilha 5; Empilha 2; Empilha 3; Empilha 4; Oper OpSoma;
 Oper OpMult; Oper OpSoma]
```

O resultado da interpretação de uma expressão `e` deve ser igual ao resultado no topo da pilha após executar o programa compilado a partir de `e`:

```
# eval @@ Soma (Const 5, Const 3);;
- : int = 8

# exec_prog @@ compila @@ Soma (Const 5, Const 3);;
- : int list = [8]

# eval e2;;
- : int = 19

# e2 |> compila |> exec_prog;;
- : int list = [19]
```

6.5 OTIMIZAÇÃO

Em um compilador, *otimização* é o processo de alterar o código gerado para torná-lo mais eficiente no uso de algum recurso, sem alterar a função original do programa. O caso mais comum é otimizar para tornar o código mais rápido de executar. Em um compilador real, a otimização pode ser feita em vários níveis e representações intermediárias.

Aqui vamos ver um exemplo de otimização baseada na simplificação da árvore sintática abstrata. Normalmente, o módulo de otimização em um compilador é organizado como um conjunto de *passagens*, cada uma realizando uma forma diferente de otimização. Cada passagem torna o código mais eficiente ou, no pior caso, não faz nada.

Uma simplificação que podemos fazer em expressões é eliminar somas em que um dos operandos é zero. Sabemos que o valor de uma expressão não é alterado quando somada a zero. A função `elimina_soma_0` faz uma travessia da árvore sintática de uma expressão, eliminando somas nas quais um dos operandos é a constante zero. Os nós que não contêm uma soma com zero são mantidos da mesma forma:

```
let rec elimina_soma_0 e =
  match e with
  | Const _ -> e
  | Soma (Const 0, e2) -> elimina_soma_0 e2
  | Soma (e1, Const 0) -> elimina_soma_0 e1
  | Soma (e1, e2) -> Soma (elimina_soma_0 e1, elimina_soma_0 e2)
  | Sub (e1, e2)  -> Sub (elimina_soma_0 e1, elimina_soma_0 e2)
  | Mult (e1, e2) -> Mult (elimina_soma_0 e1, elimina_soma_0 e2)
```

Podemos ver essa simplificação em ação:

```
# let e1 = Mult (Soma (Const 0, Const 5),
                 Soma (Const 3, Soma (Const 4, Const 0)));;
val e1 : exp =
  Mult (Soma (Const 0, Const 5),
        Soma (Const 3, Soma (Const 4, Const 0)))

# print e1;;
- : string = "((0 + 5) * (3 + (4 + 0)))"

# print @@ elimina_soma_0 e1;;
- : string = "(5 * (3 + 4))"
```

Da mesma forma, poderíamos facilmente criar outras passagens de otimização, por exemplo, para eliminar multiplicação por um. Como essas simplificações são aplicadas na árvore sintática abstrata, elas podem ser usadas no compilador, antes de traduzir a expressão para código da máquina de pilha, ou mesmo no interpretador, antes de executar a expressão.

Essas simplificações são independentes da plataforma de execução. Outras otimizações poderiam ser feitas no código da máquina de pilha, exemplificando as otimizações dependentes de

plataforma.

Neste capítulo, vimos um conjunto de programas puramente funcionais que fazem interpretação e compilação de uma linguagem de expressões, e simulação de uma máquina de pilha. Entretanto, estes só são funcionais no REPL, porque ler o conteúdo de um arquivo ou escrever o resultado da compilação em outro arquivo são operações imperativas. No próximo capítulo, vamos sair do universo puramente funcional e aprender como usar as características imperativas da linguagem OCaml.

Capítulo 7

CARACTERÍSTICAS IMPERATIVAS

Nesse ponto, já vimos todo o núcleo funcional da linguagem OCaml, e vimos que é possível fazer muito com esse núcleo funcional. Por isso, programadores OCaml tendem a usar código funcional sempre que possível. Como já discutimos no capítulo *Introdução*, código sem efeitos colaterais é mais fácil de entender e de manter, além de ser mais propício para a composição dos componentes do programa em novas combinações.

Mas um programa útil não pode existir sem efeitos colaterais. Até agora, só fomos capazes de ver o resultado dos programas funcionais puros, porque o REPL mostra o valor das expressões avaliadas, então, os efeitos ficam isolados no REPL. Mas para um programa compilado, independente do REPL, teríamos de chamar uma função de impressão para ver qualquer resultado. Isso já é um efeito.

A linguagem OCaml permite o uso de efeitos de forma similar às linguagens imperativas, o que difere das chamadas linguagens funcionais *puras* como Haskell, que controlam o uso de efeitos através dos tipos (usando o mecanismo de *mônadas*). Embora em OCaml seja possível usar os efeitos diretamente, em alguns casos a sintaxe do código imperativo na linguagem é menos prática.

Por exemplo, as variáveis mutáveis (referências), que veremos

neste capítulo, usam uma sintaxe menos direta que a usada para as variáveis imutáveis. Isso serve como um incentivo para preferir o uso de variáveis imutáveis, juntamente com funções recursivas, em vez de variáveis mutáveis com *loops*.

Contudo, existem casos em que a melhor solução é usar código imperativo, e isso é possível em OCaml. A regra geral a usar deve ser *programação funcional sempre que possível, e imperativa quando necessário*.

Neste capítulo, veremos as características imperativas da linguagem OCaml, incluindo operações de entrada e saída, variáveis e registros mutáveis, *arrays* (que são mutáveis por padrão em OCaml) e exceções.

7.1 O TIPO UNIT

Como já vimos antes, OCaml é uma linguagem orientada a expressões, o que significa que, exceto por definições, a linguagem básica contém apenas expressões. Os *comandos* encontrados em linguagens imperativas não existem em OCaml. Isso funciona bem para a parte puramente funcional, já que cada expressão e subexpressão possuem um valor.

Quando incluímos características imperativas na linguagem, surge a pergunta: qual deve ser o valor retornado por uma função que imprime algo na tela? Programação imperativa envolve comandos que são executados pelos efeitos colaterais que causam, e não pelo valor que retornam. Para que a linguagem continue sendo orientada a expressões, é preciso que essas partes imperativas retornem um valor.

Para isso, existe o tipo *unit*, que é um tipo em OCaml que só possui um valor, designado pelo literal `()`:

```
# ();;
- : unit = ()
```

Como o tipo *unit* só possui um valor, este não carrega nenhum conteúdo de informação (como pode ser visto na Teoria da Informação, o conteúdo de informação de uma fonte que sempre transmite o mesmo sinal é zero). Expressões imperativas, que não possuem nenhum valor válido de retorno, retornam `()`, portanto, têm tipo de retorno `unit`.

Por exemplo, a função `print_string` da biblioteca padrão imprime uma *string* e tem o seguinte tipo:

```
# print_string;;
- : string -> unit = <fun>
```

Dada uma *string*, a função `print_string` a imprime na tela, que é o efeito colateral desejado, e retorna `()`:

```
# print_string "cellar door ";;
cellar door - : unit = ()
```

A maioria das expressões imperativas que veremos neste capítulo retorna `unit`.

7.2 ENTRADA E SAÍDA

Um tipo comum de efeito imperativo são as funções de entrada e saída, que servem para se comunicar com dispositivos como a tela, o teclado e os discos.

Fora do REPL, quase todo programa usa funções de entrada e saída. Para imprimir na tela, existem algumas funções simples que dependem do tipo que será impresso:

- `print_int` — Imprime um inteiro;
- `print_float` — Imprime um número de ponto-flutuante;

- `print_string` — Imprime uma *string*;
- `print_endline` — Também imprime uma *string*, mas adiciona um caractere de nova linha ao final.

Mas geralmente convém mais usar as funções mais elaboradas do módulo `Printf`. Esse módulo contém funções similares às da família `printf` da linguagem C. A função `Printf.printf` é o caso mais simples: ela recebe uma *string* de formato seguida por valores que serão formatados e impressos de acordo com o formato:

```
# Printf.printf "Inteiro: %d, float: %5.3f\n" 42 3.14159265;;
Inteiro: 42, float: 3.142
- : unit = ()
```

É importante notar que, diferente do que acontece em C, a função `Printf.printf` em OCaml verifica se o tipo dos valores corresponde ao que está no formato:

```
# Printf.printf "Inteiro: %d, float: %d\n" 42 3.14159265;;
Error: This expression has type float but an expression was
expected of type int
```

Nesse último exemplo, a *string* de formato especifica que o segundo valor é um inteiro `%d`, mas o que é passado é um número de ponto-flutuante, daí o erro mostrado.

Outras variantes de `printf` muito usadas no módulo `Printf` são: `Printf.sprintf`, que coloca o resultado da formatação em uma *string* em vez de imprimir na tela; e `Printf.fprintf`, que escreve o resultado da formatação em um arquivo. De forma similar, o módulo `Scanf` contém funções da família `scanf` para ler valores de acordo com uma *string* de formato, similares às funções de mesmo nome na linguagem C. Mas ainda não vimos como usar arquivos a partir de OCaml.

Trabalhando com arquivos

Na biblioteca padrão existem dois tipos que representam

arquivos (e outros dispositivos): `in_channel` representa dispositivos de entrada, como arquivos abertos em modo de leitura; e `out_channel` representa dispositivos de saída, incluindo arquivos abertos em modo de escrita. Os canais predefinidos `stdin`, `stdout` e `stderr` representam a entrada padrão, saída padrão e saída de erro padrão, respectivamente.

A função `open_in` recebe um nome de arquivo e tenta abrir o arquivo correspondente em modo de leitura, retornando um `in_channel`. Para saída, a função `open_out` recebe um nome de arquivo e tenta abrir o arquivo correspondente em modo de leitura, retornando um `out_channel` se bem- sucedida.

Para sistemas que diferenciam arquivos de texto e arquivos binários, existem funções correspondentes `open_in_bin` e `open_out_bin` que abrem arquivos binários para leitura e escrita, respectivamente (em sistemas Unix, não existe diferença entre modo binário e modo texto). Também existem as funções `open_in_gen` e `open_out_gen`, que permitem especificar mais opções e as permissões de arquivo. Para mais detalhes, consultar a documentação em http://caml.inria.fr/pub/docs/manual-ocaml-4.01/libref/Pervasives.html#TYPEin_channel.

Uma vez de posse de um `in_channel`, podemos ler do canal usando funções como `input_char` para ler um único caractere, `input_line` para ler uma linha de texto, e `input` para ler uma sequência de *bytes*. Também podemos usar `Scanf.fscanf` para ler de um `in_channel` segundo um formato.

Da mesma forma, com um `out_channel`, podemos escrever no canal usando `output_char` para escrever um caractere, `output_string` para escrever uma *string* e e `output_bytes` para escrever uma sequência de *bytes*. Também podemos usar `Printf.fprintf` para escrever em um `out_channel` de acordo com uma *string* de formato.

Uma vez terminado o trabalho com um arquivo, deve-se fechá-lo usando `close_in` para arquivos de leitura e `close_out` para arquivos de escrita.

7.3 SEQUENCIAMENTO DE EXPRESSÕES

Como vimos, código imperativo com frequência inclui expressões que são avaliadas apenas pelo efeito colateral que causam, e não por seu resultado (elas retornam `unit`). Quando é necessário executar várias expressões imperativas em sequência, podemos usar o operador de sequenciamento de expressões.

Em OCaml, esse operador é o ponto e vírgula `;`. Uma expressão `e1; e2; ...; en` avalia as expressões `e1`, depois `e2` etc., nesta sequência, ignorando os resultados de todas, menos da última, `en`. O resultado de `en` é o resultado de toda a sequência.

O caso ideal é que todas as expressões exceto a última retornem `unit`, e o compilador vai mostrar um aviso caso uma expressão com resultado de tipo diferente de `unit` seja colocada em uma sequência em uma posição anterior à última.

Em alguns contextos, é possível usar o sequenciamento de expressões diretamente, apenas colocando ponto e vírgula entre elas. Por exemplo, uma situação que pode ser útil é inserir expressões de impressão em uma função para ver resultados intermediários e depurar o seu funcionamento (essa não é a melhor forma de depurar código, mas serve aqui como exemplo). Usando um exemplo já visto no capítulo *Tipos e valores básicos*, uma função que calcula a raiz positiva de uma equação de segundo grau (se existir), vamos inserir expressões de impressão para ver o valor do delta:

```
let raiz_positiva a b c =
  Printf.printf "Na funcao raiz_positiva\n";
```

```
  let delta = b *. b -. 4.0 *. a *. c in
  Printf.printf "Delta = %5.3f\n" delta;
  if delta >= 0.0 then
    (-.b +. sqrt delta) /. 2.0 *. a
  else infinity
```

Essa função imprime duas mensagens, uma sinalizando o começo da execução da função, e outra informando o valor do delta calculado:

```
# raiz_positiva 3.0 4.0 5.0;;
Na funcao raiz_positiva
Delta = -44.000
- : float = infinity
```

O resultado da função é a última expressão na sequência, que é o `if`. Sintaticamente, a estrutura da função é `e1; (let v = e2 in e3; e4)`, ou seja, uma sequência de duas expressões, na qual a segunda é um `let .. in` cujo corpo é uma sequência de duas expressões. O resultado da expressão completa é o resultado do `let .. in`, que por sua vez é dado pelo valor do `if`.

Em alguns contextos, principalmente dentro de outras estruturas de controle como a expressão condicional `if`, uma sequência de expressões precisa estar contida entre parênteses para evitar ambiguidade. Como a sequência é normalmente usada com expressões imperativas, existe uma sintaxe alternativa para uma expressão entre parênteses: em vez de escrever `( e1; e2; ...; en )`, pode-se usar `begin e1; e2; ...; en end`, trocando os parênteses por `begin` e `end`. Não há nenhuma diferença entre as duas formas, mas a convenção recomendada é usar `begin` e `end` quando a sequência de expressões estiver dentro de outra estrutura de controle.

Por exemplo, se quisermos inserir um `printf` no cálculo da raiz, a ambiguidade sintática resultante causa um erro:

```
# let raiz_positiva a b c =
    let delta = b *. b -. 4.0 *. a *. c in
```

```
    if delta >= 0.0 then
      Printf.printf "Existe raiz positiva\n";
      (-.b +. sqrt delta) /. 2.0 *. a
    else infinity;;
Error: Syntax error
```

O REPL indica o erro no `else`, pois aparentemente esse `else` está ligado à segunda expressão da sequência, na qual não existe nenhum `if`. A forma correta de fazer isso é protegendo a sequência usando parênteses, ou `begin` e `end`. Como a sequência está em uma estrutura de controle, recomenda-se a segunda forma:

```
# let raiz_positiva a b c =
    let delta = b *. b -. 4.0 *. a *. c in
    if delta >= 0.0 then
      begin
        Printf.printf "Existe raiz positiva\n";
        (-.b +. sqrt delta) /. 2.0 *. a
      end
    else infinity;;
val raiz_positiva : float -> float -> float -> float = <fun>
```

Erros de sintaxe em código usando sequências de expressões geralmente podem ser resolvidos colocando `begin` e `end` em torno da sequência. O uso de parênteses ou `begin .. end` também pode ajudar a evitar outros tipos de ambiguidade sintática, como às vezes acontece com o uso de um `match` dentro de um caso de outro `match`, por exemplo.

7.4 ATUALIZAÇÃO FUNCIONAL DE REGISTROS

Registros em OCaml podem ser atualizados de maneira funcional, criando um registro similar, mas com o valor de um registro alterado. Essa não é uma característica imperativa da linguagem, porém é interessante contrastar a atualização funcional com a atualização imperativa que será vista a seguir.

Vamos usar novamente o exemplo de pontos em um plano, com

a seguinte definição de tipo:

```
type ponto2d = { x : int; y : int }
```

Uma função que move um ponto verticalmente pode ser escrita da seguinte forma:

```
let mover_vert p delta = { x = p.x; y = p.y + delta }
```

Fica claro pela sintaxe que a função `mover_vert` cria um novo ponto cujo componente `x` é igual ao ponto passado, mas cujo componente `y` tem o valor adicionado ao `delta`. Essa é uma atualização funcional, que não altera o objeto original:

```
# let p1 = {x = 2; y = 6};;
val p1 : ponto2d = {x = 2; y = 6}

# let p2 = mover_vert p1 10;;
val p2 : ponto2d = {x = 2; y = 16}

# p1;;
- : ponto2d = {x = 2; y = 6}
```

No caso do tipo `ponto2d`, que só tem dois campos, não é difícil criar uma função de atualização. Se o registro tiver vários campos, fica tedioso copiar todos os que não são alterados. Por isso, existe uma sintaxe especial para atualização de registros. Basta colocar, entre chaves, o valor de registro de base e adicionar a atualização com a palavra-chave `with`, ou seja, `{ p with ... }`, em que `p` é o registro original e no lugar das reticências ficam os campos que são atualizados.

Por exemplo, reescrevendo a função `mover_vert` usando essa sintaxe:

```
let mover_vert p delta = { p with y = p.y + delta }
```

Nesse caso, não houve muita economia, pois só existem dois campos no registro, mas caso `p` fosse um registro com 10 campos, a função seria ainda a mesma. Vários campos podem ser atualizados

ao mesmo tempo, separando as atualizações com ponto e vírgula:

```
# { p1 with x = 2; y = 7 };;
Characters 0-24:
Warning 23: all the fields are explicitly listed in this record:
the 'with' clause is useless.
- : ponto2d = {x = 2; y = 7}
```

Nesse caso, atualizar os dois campos é o mesmo que criar um registro completamente novo; o *warning* indica que a sintaxe de atualização é inútil.

7.5 REGISTROS COM CAMPOS MUTÁVEIS

Variáveis em OCaml são imutáveis, como já vimos. Mas componentes de um valor podem ser alterados em alguns casos. Um desses casos é que é possível marcar campos de um registro como mutáveis.

Atualizações funcionais de registros não têm efeitos colaterais e são interessantes quando é preciso manter o valor original antes da alteração. Mas existem casos em que é mais eficiente alterar o valor de um campo em um registro de maneira imperativa, sem criar uma cópia do registro original. Nesses casos, podem ser usados campos mutáveis.

É preciso marcar o campo como mutável na declaração do tipo do registro, usando a palavra-chave `mutable`. Por exemplo, um registro para pontos no plano com campos mutáveis é declarado da seguinte forma:

```
type ponto2d = { mutable x: int; mutable y: int }
```

Campos mutáveis podem ter seu valor atualizado usando o operador `<-` na forma `r.c <- v`, em que `r` é o registro, `c` é o campo, e `v` é o novo valor. A expressão de atualização de valor retorna `()`:

```
# let p1 = {x = 5; y = 7};;
val p1 : ponto2d = {x = 5; y = 7}

# p1.x <- 20;;
- : unit = ()
```

Uma versão imperativa de `mover_vert` seria:

```
let mover_vert_imp p delta = p.y <- p.y + delta
```

Note a diferença no tipo das duas funções:

```
# mover_vert;;
- : ponto2d -> int -> ponto2d = <fun>

# mover_vert_imp;;
- : ponto2d -> int -> unit = <fun>
```

7.6 REFERÊNCIAS

Referências são células ou variáveis mutáveis. Isso não muda o que foi dito antes: as variáveis em OCaml são sempre imutáveis. Variáveis mutáveis na verdade são açúcar sintático para registros de apenas um campo mutável, e são diferenciadas das variáveis normais em OCaml por terem sintaxe e tipos diferentes.

Podemos criar uma referência usando a função `ref`. Para obter o valor de uma referência, é preciso usar o operador `!` e, para alterar o valor de uma referência, deve-se usar o operador de atribuição `:=`:

```
# let xref = ref 0;;
val xref : int ref = {contents = 0}

 # !xref;;
- : int = 0

# xref := 5;;
- : unit = ()

 # !xref;;
- : int = 5
```

Já vemos, pela resposta do REPL na criação da referência `xref`, o que realmente é uma referência: um registro com apenas um campo mutável chamado `contents`. A definição do tipo para referências e a função de criação `ref` podem ser escritos da seguinte forma:

```
type 'a ref = { mutable contents: a }

let ref x = { contents = x }
```

O tipo e a função ambos se chamam `ref`, e isso não é um problema em OCaml, pois tipos e valores pertencem a espaços de nomes diferentes. Os operadores `!` e `:=` também não são especiais, e podem ser definidos na própria linguagem:

```
let ( ! ) r = r.contents

let ( := ) r v = r.contents <- v
```

Uma consequência de as variáveis mutáveis usarem uma sintaxe diferente das imutáveis é que não é simples mudar uma variável mutável para imutável em um programa, pois isso exige mudanças em todos os pontos onde ela é utilizada. Isso funciona como uma forma de desestimular o uso de variáveis mutáveis exceto quando realmente necessário.

O tipo de uma referência também é diferente do tipo de uma variável equivalente. Enquanto uma variável inteira tem tipo `int`, uma referência inteira tem tipo `int ref`.

7.7 ARRAYS

Uma das estruturas de dados mais comuns em programas imperativos é o vetor ou *array*. Uma parte considerável do código C existente se resume a processar *arrays*, geralmente em *loops*.

Apesar de serem ambos arranjos lineares de elementos, as listas

e os *arrays* possuem diferenças importantes. As listas têm tamanho variável, são estruturas eficientes para inserir ou remover elementos na frente, mas o tempo para acessar um elemento específico da lista é proporcional à posição desse elemento na lista. Um *array*, por sua vez, tem tamanho fixo; em um *array* não é eficiente inserir ou remover elementos em qualquer posição, mas acessar elementos em qualquer posição é eficiente.

Arrays normalmente são estruturas de dados imperativas, nas quais é possível alterar o valor de qualquer elemento. Já as listas em OCaml são estruturas puramente funcionais, em que é impossível alterar o valor de um elemento diretamente (embora isso seja possível recorrendo a truques de baixo nível).

As funções para trabalhar com *arrays* na biblioteca padrão estão localizadas no módulo `Array`. Para criar um *array*, pode-se usar a função `Array.make`, especificando o tamanho do *array* e qual o valor inicial dos elementos. O código a seguir cria um *array* com 10 elementos inicializados com o valor zero:

```
# Array.make 10 0;;
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Como podemos ver na resposta do REPL no exemplo anterior, existe uma sintaxe para literais de *array*, similar à sintaxe para listas, mas usando `[|` e `|]` como delimitadores. Para *arrays* pequenos, podemos criá-los diretamente com essa sintaxe:

```
# let a1 = [| 4; 8; 15; 16; 23; 42 |];;
val a1 : int array = [|4; 8; 15; 16; 23; 42|]
```

Para acessar o elemento de índice `i` em um *array* `a`, usa-se a sintaxe `a.(i)`, com os índices começando a partir do zero, como nas *strings*:

```
# a1.(0);;
- : int = 4
```

```
# a1.(5);;
- : int = 42
```

Tentar acessar um elemento que não existe no *array* causa uma exceção:

```
# a1.(7);;
Exception: Invalid_argument "index out of bounds".
```

Embora seja possível ter *arrays* funcionais (como acontece em Haskell), a utilidade desse tipo de estrutura é limitada. Os *arrays* em OCaml são estruturas imperativas. Para alterar o valor de um componente de um *array*, podemos usar o operador `<-`, assim como acontece com campos mutáveis em registros:

```
# a1.(2) <- 51;;
- : unit = ()

# a1;;
- : int array = [|4; 8; 51; 16; 23; 42|]
```

Como acontece com outras construções imperativas, alterar o valor de um elemento de um *array* retorna `()`.

O módulo `Array` da biblioteca padrão contém várias funções úteis para trabalhar com *arrays*. Por exemplo, `Array.length` retorna o tamanho de um *array*:

```
# Array.length a1;;
- : int = 6
```

Funções de alta ordem com arrays

O módulo `Array` também inclui funções de alta ordem para trabalhar com *arrays*. Embora seja possível usar *loops* imperativos para processar *arrays*, como veremos em breve, usar funções de alta ordem geralmente resulta em código mais conciso.

Podemos fazer o mapeamento puramente funcional de *arrays* usando `Array.map`. O resultado é um novo *array* com os

elementos alterados de acordo com a função de mapeamento (o *array* original não é afetado):

```
 # Array.map (fun x -> x * x) a1;;
- : int array = [|16; 64; 2601; 256; 529; 1764|]

# a1;;
- : int array = [|4; 8; 51; 16; 23; 42|]
```

Uma variante de função de mapeamento que é útil com *arrays* é `Array.mapi` (também existe `List.mapi` para listas), em que a função de mapeamento recebe não só o valor, mas também o índice do elemento. Muitos programas que manipulam *arrays* em linguagens imperativas usam o valor do índice do elemento no processamento; esse valor muitas vezes vem do índice de um *loop*. Uma forma de traduzir esses *loops* imperativos para OCaml sem usar *loops* é usando `Array.mapi` .

```
# Array.mapi (fun i x -> i * x) a1;;
- : int array = [|0; 8; 102; 48; 92; 210|]
```

Para alterar o valor dos elementos do *array* diretamente em vez de criar outro *array*, podemos usar as funções `Array.iter` e `Array.iteri` . Ambas recebem uma função imperativa que causam algum efeito para cada elemento do *array*, sendo que na segunda a função imperativa também recebe o índice do elemento.

A função imperativa pode causar qualquer efeito, inclusive alterar o valor dos elementos do *array* usando o operador `<-` . Por exemplo:

```
 # a1;;
- : int array = [|4; 8; 51; 16; 23; 42|]

# Array.iteri (fun i x -> a1.(i) <- x * x) a1;;
- : unit = ()

# a1;;
- : int array = [|16; 64; 2601; 256; 529; 1764|]
```

Também existem as funções `List.iter` e `List.iteri` para

listas, mas como normalmente não é possível alterar imperativamente o valor de elementos de listas, essas funções só podem ser usadas para outros efeitos.

As funções `Array.fold_left` e `Array.fold_right` estão disponíveis para fazer redução em *arrays*, e funcionam da mesma forma que para listas.

7.8 ESTRUTURAS DE CONTROLE IMPERATIVAS

A linguagem OCaml também inclui algumas estruturas de controle imperativas que são familiares para programadores imperativos e que são convenientes para os casos em que expressar o programa imperativamente é a melhor escolha (ou para fazer uma primeira versão de código portado de alguma linguagem imperativa).

Uma dessas estruturas é o chamado "`if` sem `else`", um `if` com apenas uma expressão após o `then`. Esse caso é um açúcar sintático simples: `if c then e` é equivalente a `if c then e else ()`, o que significa que a expressão `e` após o `then` deve ter tipo `unit`.

Voltando à função para calcular a raiz positiva de uma equação de segundo grau, vamos imprimir uma mensagem se o valor do delta calculado for igual a zero. Se o delta não for zero, nada deve ser impresso:

```
let raiz_positiva a b c =
  let delta = b *. b -. 4.0 *. a *. c in
  if delta = 0.0 then Printf.printf "delta = 0, raiz unica\n";
  if delta >= 0.0 then
    (-.b +. sqrt delta) /. 2.0 *. a
  else infinity
```

Como o `if` sem `else` retorna `()`, é possível usar essa

construção como parte de uma sequência.

Loops

As estruturas de controle imperativas `for` e `while` podem ser usadas em OCaml. Isso pode ser útil em algumas situações, por exemplo, como forma inicial de portar código escrito em uma linguagem imperativa.

Um exemplo é uma função para calcular a soma de todos os elementos de um `array`. Usando um `for`, podemos escrevê-la da seguinte forma:

```
let soma_array a =
  let soma = ref 0 in
  let tam = Array.length a in
  for i = 0 to (tam - 1) do
    soma := !soma + a.(i)
  done;
  !soma
```

Exceto pela sintaxe, essa função seria escrita basicamente dessa maneira em muitas linguagens imperativas como a linguagem C: uma variável acumulando o resultado da soma, e um *loop* varrendo todos os elementos do *array* e somando-os ao acumulador. Apesar de ser um estilo de código familiar para programadores de linguagens imperativas, é um estilo de programação de baixo nível, em que é necessário lidar com índices, acumuladores, condições de terminação etc.

Como vimos, a mesma função pode ser escrita de maneira muito mais compacta e em alto nível usando uma função de alta ordem:

```
let soma_array a =
  Array.fold_left (+) 0 a
```

Os *loops* usando `for` podem acontecer em duas formas:

- `for i = v1 to v2 do e done` avalia a expressão `e` um total de `(v2-v1+1)` vezes, com `i` assumindo valores crescentes de `v1` até `v2`, inclusive.
- `for i = v1 downto v2 do e done` avalia a expressão `e` com `i` assumindo valores decrescentes de `v1` até `v2`.

Um `for` usando `downto` funciona da mesma forma, mas contando de forma decrescente:

```
# for i = 5 downto 1 do
    Printf.printf "iteracao %d\n" i
  done;;
iteracao 5
iteracao 4
iteracao 3
iteracao 2
iteracao 1
- : unit = ()
```

Não existe uma forma de alterar o incremento em um `for`, que é sempre 1 para um `for .. to` e -1 para um `for .. downto`. Se for necessário usar um incremento diferente, deve-se procurar outra forma de iteração (funções de alta ordem, recursividade etc.).

A outra forma de *loop* imperativo em OCaml é o `while`, cuja forma geral é `while c do e done`, com `c` uma expressão booleana e `e` uma expressão. Essa forma avalia `e` repetidamente enquanto a condição `c` tiver valor `true`.

7.9 EXCEÇÕES

As exceções são um mecanismo de controle frequentemente usado para tratamento de erros (*situações excepcionais*, daí o nome). Elas permitem tratar erros não localmente caso seja necessário, ou seja, o código que trata o erro pode estar sintaticamente longe do local onde o erro ocorreu.

Exceções são uma característica imperativa, pois a transferência não local de controle é um efeito colateral que afeta a ordem natural de avaliação das expressões.

Exceções são valores do tipo `exn`. Algumas exceções já são definidas pela biblioteca padrão da linguagem, por exemplo, a exceção `Invalid_argument`, usada para sinalizar que um argumento inválido foi passado para uma função. `Invalid_argument` é um construtor de exceção que recebe um parâmetro: uma mensagem de erro.

```
# Invalid_argument "mensagem";;
- : exn = Invalid_argument "mensagem"
```

Para disparar uma exceção, é preciso usar a função `raise`:

```
# raise @@ Invalid_argument "mensagem";;
Exception: Invalid_argument "mensagem".
```

Por exemplo, a função `sqrt` da biblioteca padrão retorna `nan` (o valor NaN, *Not a Number*, designado no padrão IEEE 754 para números de ponto flutuante) quando se tenta obter a raiz quadrada de um número negativo:

```
# sqrt @@ -.2.3;;
- : float = nan
```

Vamos escrever uma função de cálculo de raiz quadrada que dispara uma exceção `Invalid_argument` quando recebe um número negativo como argumento.

```
# let sqrt_exn x =
    if x < 0.0 then
      raise @@ Invalid_argument "sqrt_exn: numero negativo"
    else
      sqrt x;;
val sqrt_exn : float -> float = <fun>
```

Essa função funciona da forma esperada:

```
# sqrt_exn 25.0;;
- : float = 5.
```

```
# sqrt_exn @@ -.2.3;;
Exception: Invalid_argument "sqrt_exn: numero negativo".
```

Mas olhando o código da função `sqrt_exn`, uma coisa parece estranha: no `if` que testa se o argumento é negativo, a expressão após o `else` tem tipo `float`, mas a expressão após o `then` dispara uma exceção. Já vimos que as duas expressões de uma expressão condicional têm de ter o mesmo tipo, e de fato é isso que acontece aqui, embora não pareça. Vamos verificar o tipo da função `raise`:

```
# raise;;
- : exn -> 'a = <fun>
```

O tipo é `exn -> 'a`, ou seja, dada uma exceção, `raise` retorna um tipo qualquer (representado pela variável de tipo `'a`). Como essa variável é polimórfica, na função `sqrt_exn` ela assume o mesmo tipo da outra expressão do `if`. A questão é: por que `raise` retorna qualquer tipo?

A chave para entender isso é que a função `raise` nunca retorna, pelo menos não para o ponto onde foi chamada. Por isso, o tipo de retorno é flexível e pode ser qualquer um.

A função `invalid_arg` é um atalho para disparar uma exceção `Invalid_argument`. Dada uma *string* de entrada, `invalid_arg` dispara uma exceção `Invalid_argument` com essa *string* como mensagem. Reescrevendo `sqrt_exn`:

```
# let sqrt_exn x =
    if x < 0.0 then
      invalid_arg "sqrt_exn: numero negativo"
    else
      sqrt x;;
val sqrt_exn : float -> float = <fun>
```

Outra exceção predefinida é `Failure`, que também recebe uma mensagem como parâmetro. A função `failwith` é um sinônimo

para a expressão `raise @@ Failure`. A exceção `Failure` é usada quando o código encontra alguma falha da qual não é possível recuperar localmente.

A exceção predefinida `Exit` existe para usar como uma saída prematura do programa. A ideia é que um programa pode disparar essa exceção em situações críticas das quais não seja possível recuperar; a exceção não deve ser tratada e causa a terminação do processo.

Tratamento de exceções

Para capturar exceções disparadas por uma expressão, é preciso usar a estrutura `try e with ...`. Se a expressão `e` dispara uma exceção, ela é testada de acordo com os padrões que ocorrem após o `with`, de forma similar ao que acontece em um `match`. Por exemplo:

```
# try
    sqrt_exn @@ -.3.7
  with
    Invalid_argument m ->
      Printf.printf "Excecao: argumento negativo\n"; 0.0;;
Excecao: argumento negativo
- : float = 0.
```

Esse código chama `sqrt_exn` com um argumento negativo, mas captura a exceção, imprime uma mensagem e retorna o valor `0.0`. O argumento da exceção pode ser usado:

```
# try
    sqrt_exn @@ -.3.7
  with
    Invalid_argument m -> Printf.printf "Excecao: %s\n" m; 0.0;;
Excecao: sqrt_exn: numero negativo
- : float = 0.
```

Exceções que não satisfazem os padrões após o `with` não são capturadas e se propagam como se não houvesse um `try`.

```
# try
    sqrt_exn @@ -.3.14
  with
    Failure _ -> Printf.printf "Falha\n"; 0.0;;
Exception: Invalid_argument "sqrt_exn: numero negativo".
```

Podemos testar a exceção com relação a vários padrões:

```
# try
    sqrt_exn @@ -.3.14
  with
    Failure _ -> Printf.printf "Falha\n"; 0.0
  | Invalid_argument _ -> Printf.printf "inv_arg\n"; 0.0;;
inv_arg
- : float = 0.
```

Também podemos usar variáveis em um padrão para capturar qualquer exceção:

```
# try
    sqrt_exn @@ -.3.14
  with
    e -> Printf.printf "Excecao\n"; 0.0;;
Excecao
- : float = 0.
```

Se uma exceção se propaga sem ser tratada no REPL, a avaliação da expressão atual é abortada e uma mensagem é impressa, como visto nos exemplos anteriores. Já em um programa compilado, uma exceção não tratada causa a terminação do processo.

Declarando novas exceções

Além das exceções já existentes na biblioteca padrão, é possível criar novas exceções para situações específicas. A forma de fazer isso é usando a palavra-chave `exception` .

```
# exception AlertaVermelho;;
exception AlertaVermelho

# raise AlertaVermelho;;
Exception: AlertaVermelho.
```

Nomes de exceção devem começar com letras maiúsculas, com

os caracteres seguintes podendo ser letras maiúsculas ou minúsculas, dígitos e *underscore*. Uma exceção pode carregar um ou mais valores, assim como um construtor de um tipo variante:

```
# exception Mensagem of string;;
exception Mensagem of string

# raise @@ Mensagem "ALERTA!!!!";;
Exception: Mensagem "ALERTA!!!!".
```

A semelhança com variantes não é por acaso: o tipo `exn` funciona como um ADT cujas variantes são as exceções. Uma diferença para os tipos variantes vistos até agora é que o tipo `exn` é *aberto*, pois novas variantes podem ser adicionadas em outros momentos, não apenas na declaração inicial do tipo. Quando usamos `exception` para declarar uma nova exceção, estamos basicamente criando uma nova variante para o tipo `exn`.

As características imperativas da linguagem OCaml a afastam de um ideal de pureza, mas a tornam mais pragmática na forma de lidar com situações que exigem o uso de efeitos colaterais. Essa forma híbrida de lidar com efeitos usando diretamente características funcionais, como ocorre em OCaml, com certeza não é a única possibilidade (Haskell usa mônadas, Clean usa um conceito similar ao de tipos lineares), mas é uma solução razoável e prática para a questão, já que não é possível criar um programa minimamente útil sem efeitos colaterais.

Capítulo 8

MÓDULOS

Todos os programas em OCaml são organizados em módulos. Até agora temos visto programas que usam apenas um módulo, e por isso não precisamos aprender como trabalhar com módulos explicitamente. Para programas maiores, o uso dos módulos ajuda a organizar os seus componentes, possibilitando a delimitação de interfaces que ajudam a desacoplar as partes.

Os módulos em OCaml também permitem novas formas de abstração que são impossíveis sem eles. Neste capítulo, vamos examinar o sistema de módulos da linguagem OCaml.

As partes mais avançadas desse sistema (funtores, restrições de compartilhamento etc.) são frequentemente consideradas entre as partes mais difíceis da linguagem. Às vezes, é necessário um tempo de experiência com OCaml para realmente entender essas características.

O leitor não deve se preocupar se não entender tudo neste capítulo imediatamente, o mais importante é saber usar o básico dos funtores, pois eles são usados na biblioteca padrão da linguagem e em outras bibliotecas importantes. Eventualmente, para quem pretende criar projetos OCaml não-triviais, é importante entender o sistema de módulos mais a fundo.

8.1 ESTRUTURAS E ASSINATURAS

Um módulo em OCaml sempre tem duas partes: a *estrutura* e a *assinatura*. A assinatura é a interface do módulo, e a estrutura é a implementação. O uso de assinaturas permite criar interfaces que escondem informações e detalhes da implementação que devem permanecer internos.

Uma estrutura é criada usando `struct .. end`. Todas as declarações (tipos, variáveis, funções, outros módulos) colocadas entre `struct` e `end` pertencem a ela. Geralmente, uma estrutura é definida para criar um módulo; a forma mais comum de uso é:

```
module Mod =
struct
  (* declarações *)
end
```

Isso declara o módulo `Mod` com a estrutura especificada. Todo módulo tem uma estrutura e uma assinatura. Se nenhuma assinatura for especificada, é assumida uma assinatura transparente, que declara tudo que existe na estrutura. Por exemplo:

```
# module Mod =
  struct
    let x = 0
  end;;
module Mod : sig val x : int end
```

O REPL responde mostrando a assinatura do módulo definido. Uma assinatura é especificada usando `sig .. end`. Não podem existir definições de valores, apenas declarações.

Um valor é declarado usando `val`, e a declaração deve incluir o tipo do valor. Por isso, assume-se a assinatura transparente para a estrutura `struct let x = 0 end` é `sig val x : int end`, como mostrado na reposta do REPL do exemplo anterior.

Variáveis e funções são especificadas em assinaturas usando `val`. E tipos são especificados usando `type`, assim como em estruturas.

Uma assinatura também pode ser entendida como o *tipo* de um módulo. Por isso, a forma de dar nome a ela é declarando um tipo de módulo usando `module type`. Por exemplo, a assinatura do módulo `Mod` usado nos exemplos anteriores pode ser escrita como:

```
module type Mod_sig =
sig
  val x : int
end
```

A única forma de especificar a assinatura de um módulo é durante a sua definição. Por isso, se for desejado especificá-la explicitamente, é preciso defini-la antes do módulo em si. Para especificá-la, usamos a forma `M : A`, por exemplo:

```
module M : A = mod_exp
```

O `mod_exp` é uma expressão que denota uma estrutura. Vamos definir explicitamente a assinatura do módulo `Mod`:

```
# module type Mod_sig =
  sig
    val x : int
  end;;
module type Mod_sig = sig val x : int end

# module Mod : Mod_sig =
  struct
    let x = 0
  end;;
module Mod : Mod_sig
```

Se a estrutura não define um item que está declarado na assinatura, isso causa um erro. O contrário, definir itens na estrutura que não aparecem na assinatura, não é um problema, e representa a possibilidade de esconder detalhes de implementação, como veremos a seguir.

8.2 ACESSO AOS ITENS DE UM MÓDULO

Itens declarados em um módulo e que sejam visíveis segundo a assinatura podem ser acessados usando a notação `M.v`, em que `M` é o nome do módulo e `v` é o item desejado. Para o módulo `Mod` definido anteriormente:

```
# Mod.x;;
- : int = 0
```

Qualquer item definido na estrutura, mas não declarado na sua assinatura, não pode ser acessado de fora do módulo. Por exemplo:

```
# module Mod2 : Mod_sig =
  struct
    let y = 6
    let x = y * 7
  end;;
module Mod2 : Mod_sig

# Mod2.y;;
Error: Unbound value Mod2.y
```

Embora `y` exista e possa ser acessada dentro do módulo `Mod2`, ela não pode ser acessada fora do módulo, pois a assinatura `Mod_sig` não declara um valor `y`.

Nomes de tipos, construtores e mesmo nomes de campos são acessados da mesma forma. Para campos de registros, a regra de acesso aos nomes em módulos tem um resultado um tanto inesperado.

Vamos lembrar de que os nomes de campos podem ser mencionados sem usar o nome do tipo de registro ao qual os campos pertencem, e o tipo do registro pode ser determinado pelos nomes dos campos. Por isso, os nomes dos campos pertencem ao espaço de nomes do módulo onde o tipo é definido, mesmo que uma variável do tipo seja definida fora do módulo.

Por exemplo, vimos no capítulo *Registros e variantes* que, se definirmos registros para representar pontos em 2 e 3 dimensões, os

nomes dos campos (especificamente, `x` e `y`) entram em conflito, pois gostaríamos de usá-los nos registros para pontos em 2 e 3 dimensões. Para solucionar isso, vamos criar módulos contendo esses tipos, começando com pontos em duas dimensões:

```
module Ponto2D =
struct
  type t = { x : int; y : int }

  let zero () = { x = 0; y = 0 }
end
```

Agora podemos criar um ponto na origem do sistema de coordenadas usando a função `zero` (cujo único parâmetro é do tipo `unit`, já que não é preciso passar nenhuma informação para a função).

```
# let p1 = Ponto2D.zero ();;
val p1 : Ponto2D.t = {Ponto2D.x = 0; y = 0}
```

Mas para acessar os campos do registro em `p1`, precisamos usar o nome do módulo de origem:

```
# p1.Ponto2D.x;;
- : int = 0
```

Como a variável `p1` pertence ao módulo atual, não precisamos especificar um nome de módulo para ela. Entretanto, o nome de campo `x` pertence ao módulo `Ponto2D`. Isso parece estranho, mas na prática raramente vemos código dessa forma, principalmente se forem usadas as formas de *abrir* os nomes de um módulo, como veremos mais adiante.

Em versões anteriores da linguagem, uma tentativa de acessar o campo `x` do ponto usando `p1.x` resultaria apenas em um erro. Versões mais recentes usam algumas heurísticas para resolver um nome de campo quando o tipo do registro é conhecido. O compilador emite um aviso nessa ocasião:

```
# p1.x;;
```

```
Characters 3-4:
Warning 40: x was selected from type Ponto2D.t.
It is not visible in the current scope, and will not
be selected if the type becomes unknown.
- : int = 0
```

Nesse caso, a heurística funciona porque o compilador conhece o tipo da variável `p1`. O problema é que é comum, em código OCaml, confiar na inferência de tipos realizada pelo compilador (como vimos em praticamente todos os exemplos neste livro), e se for necessário inferir o tipo do registro pelo nome dos campos, o compilador não vai adivinhar de que registro vêm os campos. Os nomes dos campos devem estar acessíveis no módulo atual. É recomendável usar o nome do módulo antes do nome do campo, como no exemplo anterior (`p1.Ponto2D.x`), ou abrir o módulo localmente, como veremos adiante.

Tipos abstratos

Da mesma forma que podemos esconder valores de um módulo usando uma assinatura, também podemos esconder tipos. Mas além de esconder totalmente um tipo interno ao módulo, também é possível esconder apenas seus detalhes, tornando-o um tipo *abstrato*. Um tipo é abstrato quando sua existência é declarada na interface do módulo, mas só isso; seus detalhes são escondidos.

Vamos criar um tipo para guardar informações sobre arquivos. Para simplificar, vamos guardar apenas o nome do arquivo e se ele está aberto ou fechado. Os tipos são:

```
type estado_arq = Fechado | Aberto

type arq = { nome : string; estado : estado_arq }
```

Isso funciona para representar as situações desejadas, mas também permite representar estados ilegais. Por exemplo, criar um arquivo já no estado `Aberto`, antes de passar por uma operação de

abertura:

```
# let a = { nome = "arquivo.txt"; estado = Aberto } ;;
val a : arq = {nome = "arquivo.txt"; estado = Aberto}
```

Isso acontece porque todos os detalhes do tipo estão expostos e podem ser manipulados livremente. Usando uma assinatura podemos tornar o tipo `arq` abstrato:

```
# module type Arq_sig =
  sig
    type t
  end;;
module type Arq_sig = sig type t end

# module Arq : Arq_sig =
  struct
    type estado = Fechado | Aberto
    type t = { nome: string; estado: estado }
  end;;
module Arq : Arq_sig
```

O módulo `Arq` contém os tipos e valores associados com arquivos. O tipo principal do módulo é `t`, antes chamado de `arq`. Como `Arq.arq` ficaria redundante, é uma convenção comum em código OCaml chamar o tipo principal de um módulo de `t`.

O problema é que agora é impossível criar um valor do tipo `Arq.t`. Para resolver isso, é preciso definir funções no módulo que possam manipular valores do tipo.

```
module type Arq_sig =
sig
  type t
  val criar : string -> t
  val abrir : t -> unit
end

module Arq : Arq_sig =
struct
  type estado = Fechado | Aberto
  type t = { nome: string; mutable estado: estado }
  let criar nome = { nome; estado = Fechado }
  let abrir arq = arq.estado <- Aberto
```

```
end
```

Pela assinatura, é possível criar novos valores do tipo `Arq.t` usando a função `Arq.criar`:

```
# let a1 = Arq.criar "arquivo.txt";;
val a1 : Arq.t = <abstr>
```

O REPL não mostra o conteúdo do valor, pois ele é abstrato. A função `Arq.criar` cria um arquivo com o nome especificado, mas com o estado fechado. Como é a única forma (fora do módulo) de criar um valor do tipo, é impossível representar valores ilegais como criar um arquivo em estado aberto antes de abri-lo.

A função `Arq.abrir` abre um arquivo e muda o seu estado de acordo. Para efetuar essa mudança de estado, o campo `estado` no registro foi declarado como mutável, e `Arq.abrir` altera o valor do campo imperativamente (por isso, retorna `unit`).

```
 # Arq.abrir a1;;
- : unit = ()
```

Como não é possível inspecionar o campo `estado`, não é possível determinar se o arquivo está aberto ou não. Para isso, precisaria ser criada uma função no módulo (e declarada na assinatura), por exemplo uma função `aberto` que retorna um valor booleano indicando se o arquivo está aberto.

Abrindo um módulo

Abrir um módulo importa todos os nomes nele visíveis para o contexto atual. Isso significa que um item `x` definido no módulo `M` pode ser acessado diretamente pelo nome `x` em vez de `M.x`.

Um módulo pode ser aberto global ou localmente. Para abri-lo globalmente, deve-se usar a declaração `open` e seu nome. Usando o módulo `Arq` anterior:

```
# open Arq;;
```

```
# let a1 = criar "arquivo.txt";;
val a1 : t = <abstr>
```

Nesse exemplo, chamamos a função `criar` (em vez de `Arq.criar`) e o tipo de `a1` é mostrado como `t` (e não `Arq.t`). Mesmo abrindo o módulo `Arq`, só ficam visíveis dele as partes que são declaradas em sua assinatura (que é `Arq_sig`), portanto, o tipo `t` continua sendo abstrato. Abrir um módulo não revela seus detalhes internos.

Abrir um módulo globalmente é algo que deve ser feito com cuidado, e em poucos casos. Um módulo que dependa de vários outros e tente abrir todos eles provavelmente vai ter conflitos de nome (começando pelo nome de tipo `t`, usado por muitos módulos).

Em programas que usam primariamente uma só biblioteca, pode fazer sentido abrir o módulo principal da biblioteca globalmente. Mesmo assim, isso pode reduzir a legibilidade do código, já que não dá para saber de imediato se um nome vem do módulo atual ou de um módulo aberto.

Uma opção para obter notações mais compactas, mas sem perder o contexto, é abrir módulos localmente. Existem duas formas de fazer isso. A primeira é usando um `let` especial:

```
 # let a1 = let open Arq in criar "arquivo.txt";;
val a1 : Arq.t = <abstr>
```

A forma geral é `let open M in e`, em que `M` é um módulo e `e` é uma expressão. Todos os nomes em `e` são tratados como se `M` estivesse aberto. A outra forma de abrir localmente é envolvendo a expressão entre parênteses, usando o nome do módulo antes:

```
# let a2 = Arq.(criar "arquivo.txt");;
val a2 : Arq.t = <abstr>
```

A forma geral é `M.(e)`, mais uma vez com `M` sendo um módulo e `e` uma expressão.

Para esse caso com apenas uma expressão, abrir um módulo localmente não é muita vantagem. Mas vamos ver outro exemplo, usando outra versão do módulo `Ponto2D`:

```
module Ponto2D =
struct
  type t = { x : int; y : int }

  let zero () = { x = 0; y = 0 }

  let criar x y = { x; y }

  let ( + ) p1 p2 = { x = p1.x + p2.x; y = p1.y + p2.y }
end
```

Agora podemos definir uma função que multiplica os valores das coordenadas de um ponto da seguinte forma:

```
let mult_coords p = Ponto2D.(p.x * p.y)
```

Em `mult_coords`, não é necessário usar o nome do módulo `Ponto2D` para acessar os campos `x` e `y`. Vamos criar um ponto que é a soma de dois outros:

```
# let p1 = let open Ponto2D in
           (criar 3 4) + (criar 11 8);;
val p1 : Ponto2D.t = {Ponto2D.x = 14; y = 12}
```

Nessa expressão, o operador de soma realiza a soma de pontos, e não a soma de inteiros, já que o módulo `Ponto2D` está aberto na expressão, e esse módulo define um operador de soma.

Abrir módulos localmente nos permite escrever expressões concisas sem poluir o espaço de nomes global. Também ajudam com a redefinição de operadores, como vimos no exemplo anterior. É possível redefinir operadores sem esconder os operadores originais de forma definitiva. Sempre que possível, é recomendável abrir módulos localmente em vez de globalmente.

8.3 MÓDULOS E ARQUIVOS

Os arquivos de código-fonte OCaml usam a extensão `.ml`, e cada um define implicitamente um módulo, da seguinte forma: um arquivo `arq.ml` define um módulo chamado `Arq` (o nome do arquivo, sem a extensão, com a primeira letra mudada para maiúscula) cujo conteúdo é a estrutura do módulo. O arquivo `arq.ml` é equivalente à seguinte definição de módulo:

```
module Arq =
struct
  (* conteúdo do arquivo arq.ml *)
end
```

A assinatura do módulo `Arq` definido pelo arquivo `arq.ml` pode ser especificada no arquivo `arq.mli`. Se o arquivo `.mli` para um módulo existe, ele é equivalente à seguinte assinatura:

```
module type Arq_sig =
sig
  (* conteúdo do arquivo arq.mli *)
end
```

A assinatura associada a um arquivo `arq.mli` não recebe um nome como a estrutura do módulo em `arq.ml` (o nome `Arq_sig` no exemplo anterior é apenas para ilustrar o funcionamento). Entretanto, se existe um arquivo `arq.mli` associado a `arq.ml`, o módulo `Arq` recebe o tipo de módulo definido no `mli`:

```
module Arq : Arq_sig =
struct
  (* conteúdo do arquivo arq.ml, com assinatura em arq.mli *)
end
```

Como já vimos, se um módulo é definido sem uma assinatura explícita, o compilador infere uma assinatura transparente que expõe todas as definições (de valores e tipos) do módulo completamente, inclusive detalhes dos tipos. Para ver a assinatura inferida pelo compilador, podemos usar a opção `-i` (para mostrar

a interface). Se colocarmos o conteúdo do módulo `Ponto2D` em um arquivo com nome `ponto2D.ml`:

```
type t = { x : int; y : int }

let zero () = { x = 0; y = 0 }

let criar x y = { x; y }

let ( + ) p1 p2 = { x = p1.x + p2.x; y = p1.y + p2.y }
```

Podemos obter a interface desse módulo usando uma opção do compilador:

```
andrei$ ocamlc -i ponto2D.ml
type t = { x : int; y : int; }
val zero : unit -> t
val criar : int -> int -> t
val ( + ) : t -> t -> t
```

Um arquivo de assinatura (com extensão `mli`) é compilado para outro com extensão `cmi`. O que é gerado ao compilar um arquivo de estrutura (um módulo com extensão `ml`) depende do compilador utilizado: para o compilador de *bytecode* (`ocamlc`), cada arquivo `ml` compilado gera outro com extensão `cmo`. O compilador nativo (`ocamlopt`) gera um com extensão `cmx` e um arquivo objeto com extensão `o`.

Se um arquivo `b.ml` depende do arquivo `a.ml`, é preciso ter uma versão compilada da assinatura de `a` (`a.cmi`) para compilar `b.ml`. Isso acontece com ambos os compiladores.

Execução de código e o ponto de entrada

Um arquivo com extensão `.ml` define um módulo, e neste módulo podemos definir variáveis e funções. Mas uma pergunta que surge é: qual é o ponto de entrada do programa? Ou seja, que função é chamada para iniciar a execução? Em muitas linguagens como C, C++ e Java, existe uma função principal (geralmente chamada

`main`) que define o ponto de entrada do programa. Em OCaml, não existe uma função principal desta forma.

Todo código que aparece em um arquivo `.ml` é executado quando o programa é executado. Se existe uma definição de função no arquivo, a execução dessa definição apenas vai criar uma nova função. Mas se colocarmos um `printf` no meio do módulo (sem ser parte de nenhuma função), ele será executado quando o programa for executado. Isso significa que a função principal em OCaml é todo o código nos arquivos que não faz parte de uma definição de função.

Em um programa composto por vários arquivos `.ml`, a ordem de execução do código em cada arquivo é definida pela ordem em que eles são especificados na compilação. Por exemplo, se um programa é composto por três arquivos `a.ml`, `b.ml` e `c.ml`, e o programa for compilado com a seguinte chamada:

```
ocamlc -o prog a.ml c.ml b.ml
```

Isso gerará um executável chamado `prog`, que executa o código nos módulos `a.ml`, `c.ml` e `b.ml`, nesta ordem.

A prática comum em OCaml é ter quase todos os arquivos do programa apenas com definições de valores (variáveis e funções), módulos e funtores. Um arquivo será o módulo principal do programa, e ao final deste arquivo é incluído código que será, efetivamente, a função principal do programa. Esse código deve retornar `()`. A estrutura é:

```
(* variaveis, funcoes, modulos,
   abertura de modulos externos, etc *)

(* funcao principal *)
let () =
  Printf.printf "Programa DEEP THOUGHT\n";
  (* Chamadas de outras funcoes *)
```

O exemplo do capítulo *Exemplo: árvores de decisão* contém vários programas que seguem essa estrutura. Por exemplo, esse é o código do arquivo `arvgenero.ml`:

```
let () =
  Printf.printf "Lendo arquivo de teste e escrevendo
                 genero.csv\n";
  Titanic.classifica_arv_genero "data/test.csv" "genero.csv"
```

O módulo `Titanic` é definido em outro arquivo e inclui a função `classifica_arv_genero`, que faz todo o trabalho do programa.

Agora sabemos como criar módulos e assinaturas para esses módulos. Porém, para usar o sistema de módulos da linguagem de maneira completa, é preciso usar os funtores, que são o assunto a seguir.

8.4 FUNTORES

Um *funtor* em OCaml é uma função que recebe um módulo como parâmetro e gera um módulo como resultado. Funtores são basicamente uma forma de criar módulos que mudam seu comportamento de acordo com o módulo passado como parâmetro para o funtor.

Isso pode ser usado para injeção de dependências, para estender módulos existentes com novas funcionalidades. Também pode ser usado para criar código que é polimórfico não só com relação a um tipo, mas também com relação a um grupo de operações definidas sobre esse tipo.

Um módulo `A` pode depender de outro módulo `B` de forma fixa: basta que `A` use algum valor de `B`. Isso significa que `A` define sua funcionalidade baseado na funcionalidade de `B`. Mas às vezes queremos criar uma dependência mais dinâmica, baseada na

interface de um módulo: podemos definir certas funcionalidades para qualquer módulo que tenha uma determinada interface. Funtores servem para isso.

Como sempre, exemplos ajudam a deixar essa discussão mais concreta. Vamos começar com um caso mais simples e didático, e depois passar para algo mais realista do uso de funtores.

Nosso primeiro exemplo mostra como estender o funcionamento de módulos usando funtores. A ideia é adicionar uma função de *logging* em módulos que implementam operações de aquisição e liberação de recursos, como arquivos. O módulo estendido vai ter as mesmas operações de aquisição e liberação, mas que adicionam uma mensagem ao *log* cada vez que são chamadas.

Os módulos que serão estendidos devem seguir a interface `Arq` adiante. Chamamos as operações de aquisição e liberação do recurso de `abrir` e `fechar`, pela semelhança com essas operações em arquivos.

```
module type Arq =
sig
  type t

  val abrir : string -> t

  val fechar : t -> unit
end
```

A interface define um tipo principal `t` (o nome usual para o tipo principal de um módulo) e duas funções, `abrir` e `fechar`. Para abrir um arquivo (ou outro tipo de recurso), precisamos passar o nome ou endereço do arquivo em uma string, e um valor que representa o arquivo ou recurso é retornado.

Para fechar, precisamos passar o valor associado ao arquivo, e a função de fechamento retorna apenas `()`. Realisticamente, uma interface como essa teria de incluir outras operações além de apenas

abrir e fechar o arquivo, mas para os nossos propósitos, isso não é necessário.

Agora vamos definir o funtor `ArqLog` que adiciona o registro em *log* às operações da interface `Arq`. Vamos definir operações de *log* simples em um módulo:

```
module Log =
struct
  let registra str = Printf.printf "::: LOG: %s\n" str
end
```

Agora podemos definir `ArqLog`. Um funtor é um módulo parametrizado que recebe outro módulo. A sintaxe reflete isso:

```
# module ArqLog(A : Arq) : Arq =
  struct
    type t = A.t

    let abrir nome =
      let logstr = Printf.sprintf "abrindo arquivo %s" nome in
      Log.registra logstr;
      A.abrir nome

    let fechar arq =
      Log.registra "fechando arquivo";
      A.fechar arq
  end;;
module ArqLog : functor (A : Arq) -> Arq
```

A resposta do REPL indica que o tipo de `ArqLog` é `functor (A : Arq) -> Arq`, mostrando que é um funtor que recebe um módulo com assinatura `Arq` e gera um módulo com a mesma assinatura. Na primeira linha, a declaração do funtor é:

```
module ArqLog(A : Arq) : Arq = ...
```

Isso define `ArqLog` como um funtor que recebe como parâmetro um módulo com assinatura `Arq`, de nome `A`. Na definição de `ArqLog`, o nome de módulo `A` representa esse parâmetro. Como o resultado do funtor `ArqLog` também deve seguir a assinatura `Arq`, isso significa que o funtor precisa definir

um tipo `t` e funções `abrir` e `fechar`.

O tipo `t` em `ArqLog` é o mesmo tipo principal do módulo recebido como parâmetro, `A.t`. A função `abrir` registra no *log* antes de chamar `A.abrir`, e `fechar` faz a mesma coisa.

Agora podemos adicionar *logging* a qualquer módulo que siga a interface `Arq`. Para exemplificar, vamos criar um módulo `ArqLeitura` que manipula arquivos para leitura:

```
module ArqLeitura : Arq =
struct
  type t = { nome: string; ch: in_channel }

  let abrir nome =
    { nome; ch = open_in nome }

  let fechar t = close_in t.ch
end
```

Com esse módulo, podemos criar um outro que manipula arquivos para leitura e registra as aberturas e fechamentos de arquivos em um *log*, aplicando o funtor `ArqLog` ao módulo `ArqLeitura`:

```
# module ArqLerLog = ArqLog(ArqLeitura);;
module ArqLerLog :
sig
  type t = ArqLog(ArqLeitura).t
  val abrir : string -> t
  val fechar : t -> unit
end
```

A sintaxe para aplicação de funtores é `F(M)`, em que `F` é o funtor e `M` é o módulo passado como parâmetro. Nesse caso, os parênteses são necessários. `ArqLerLog` é uma extensão do módulo `ArqLeitura` que registra a ocorrência das operações em *log*. Qualquer outro módulo que siga a assinatura `Arq` pode ser estendido similarmente.

Conjuntos e mapas

Um exemplo típico do uso de funtores em OCaml são estruturas como conjuntos (*sets*) e mapas (*maps*). Estas muitas vezes são implementadas usando árvores binárias balanceadas (como árvores AVL ou árvores vermelho-preto), e o funcionamento dessas árvores depende de uma ordenação dos elementos. Ou seja, para inserir ou procurar elementos na árvore, é necessário comparar o valor do elemento com outros elementos na árvore.

A título de ilustração, vamos implementar uma estrutura de dados representando um conjunto. Para isso, usaremos uma árvore binária de busca não-balanceada. Usar uma estrutura de árvore balanceada como AVL seria apenas uma melhoria na eficiência e não afetaria o resto do exemplo.

Para buscar um elemento no conjunto, fazemos uma busca na árvore binária. A árvore segue a regra usual das árvores binárias de busca: cada nó guarda um valor V, e todos os valores menores que V abaixo desse nó aparecem na subárvore esquerda, enquanto os valores maiores que V aparecem na subárvore direita. Esse é o mesmo algoritmo que vimos no capítulo *Polimorfismo e mais padrões*. Para implementar esse algoritmo, precisamos ter operações que comparem dois valores na árvore.

Embora exista a operação de comparação polimórfica `compare`, ela nem sempre é adequada para qualquer tipo. Os operadores de comparação em OCaml chamam `compare`, portanto, se escrevermos uma função de inserção em conjunto da seguinte forma:

```
module Set =
struct
  type 'a t = No of 'a * 'a t * 'a t | Folha

  let rec busca s x =
    match s with
    | No (k, esq, dir) when k = x -> Some s
    | No (k, esq, dir) ->
```

```
        if x < k then busca esq x
        else busca dir x
    | Folha -> None
end
```

Isso funciona para criar conjuntos com elementos de qualquer tipo `'a`, desde que a comparação polimórfica da linguagem seja adequada para esse tipo. Infelizmente, isso nem sempre acontec. Além disso, a comparação polimórfica é fixa para cada tipo: é uma comparação baseada na estrutura do tipo.

Isso funciona bem para os tipos básicos e para muitos outros tipos, mas nem sempre é o adequado para a aplicação. Por exemplo, a comparação predefinida para listas (e *arrays*) segue uma ordem lexicográfica:

```
# [1; 2; 3] < [1; 2; 3; 4];;
- : bool = true

# [| 2; 1; 1 |] < [| 1; 7; 9 |];;
- : bool = false
```

Mas digamos que queremos usar os *arrays* como vetores, e comparar vetores pela sua magnitude. Nesse caso, não poderíamos usar a comparação polimórfica predefinida na linguagem, e não poderíamos usar o módulo `Set` definido antes. Queremos definir um módulo `Set` que possa variar não só o tipo do elemento do conjunto, mas também a função de comparação usada. Para isso, podemos usar funtores.

Primeiro, definiremos uma assinatura para módulos que incluem um tipo e uma função de comparação para esse tipo:

```
module type Comp =
sig
  type t

  val compare : t -> t -> int
end
```

Aqui seguimos a convenção da linguagem OCaml de ter uma

função de comparação que retorna: zero se os valores comparados forem iguais; -1 se o primeiro valor for menor que o segundo; e +1 se o primeiro valor for maior que o segundo. Agora podemos definir o funtor `Set`:

```
module Set(E : Comp) =
struct
  type t = No of E.t * t * t | Folha

  let vazio = Folha

  let rec busca s x =
    match s with
    | No (k, esq, dir) when E.compare k x = 0 -> Some s
    | No (k, esq, dir) ->
        if E.compare k x < 0 then busca esq x
        else busca dir x
    | Folha -> None

  let rec adiciona s x =
    match s with
    | No (k, esq, dir) when E.compare k x = 0 -> s
    | No (k, esq, dir) ->
        if E.compare k x < 0 then No (k, adiciona esq x, dir)
        else No (k, esq, adiciona dir x)
    | Folha -> No (x, Folha, Folha)

  let rec lista s =
    match s with
    | No (k, esq, dir) -> k :: (lista esq @ lista dir)
    | Folha -> []
end
```

Dado um módulo que segue a interface `Comp` (um tipo comparável), o funtor `Set` constrói um módulo para conjuntos cujos elementos são do tipo `t` do módulo passado como parâmetro. O módulo criado vai ter o valor `vazio` que representa um conjunto vazio e três operações: uma para buscar um elemento no conjunto; uma para adicionar um novo elemento (retornando um novo conjunto com o elemento adicionado — é uma operação funcional que não altera o valor original); e uma função que retorna uma lista com os elementos do conjunto.

Com esse funtor, podemos criar módulos para conjuntos que guardam elementos de tipos específicos. Por exemplo, um módulo para conjuntos de inteiros:

```
module IntComp : Comp =
struct
  type t = int
  let compare = compare
end

module IntSet = Set(IntComp)
```

`IntComp` define o tipo `t` como `int` e a função de comparação do módulo como sendo igual à função predefinida `compare` da biblioteca padrão (que é perfeitamente adequada para comparar valores inteiros).

Frequentemente, precisamos criar um módulo como `IntComp`, que existe apenas para ser passado como parâmetro para um funtor. Nesse caso, não é necessário criar um módulo nomeado, podemos defini-lo na própria chamada ao funtor. O código a seguir é (quase) equivalente ao exemplo anterior:

```
module IntSet = Set(struct
                      type t = int
                      let compare = compare
                    end)
```

Se criarmos o módulo `IntSet` com esse exemplo anterior, podemos usá-lo, por exemplo, em uma função para remover valores duplicados em uma lista:

```
let remove_dup l =
  let conj = List.fold_left IntSet.adiciona IntSet.vazio l in
  IntSet.lista conj
```

Essa função primeiro adiciona todos os elementos da lista `l` em um conjunto, usando um *fold*, depois transforma o conjunto em uma lista. A função funciona como esperado:

```
# remove_dup [1; 3; 1; 4; 5; 4; 2];;
```

```
- : int list = [1; 3; 4; 5; 2]
```

Se o módulo `IntSet` não for necessário em outros contextos, podemos criá-lo localmente.

Módulos locais

Da mesma maneira como criamos valores locais usando uma forma `let .. in` , podemos criar módulos locais para serem usados apenas em uma expressão usando a forma `let module .. in` . Com isso, podemos escrever a função `remove_dup` sem precisar criar um módulo antes:

```
let remove_dup l =
  let module ISet =
    Set(struct type t = int let compare = compare end)
  in
  let conj = List.fold_left ISet.adiciona ISet.vazio l in
  ISet.lista conj
```

Essa função funciona como a versão anterior, mas dispensa a criação de um módulo `IntSet` externo, que pode não ser necessário se nenhuma outra parte do código o utiliza. Módulos locais também podem ser usados para criar um *alias* local mais curto para um módulo, e para trabalhar com módulos de primeira classe.

Restrições de compartilhamento

Uma coisa curiosa acontece no exemplo anterior se definirmos o módulo `IntSet` da primeira forma que foi vista, através da criação de um módulo `IntComp` que implementa a assinatura `Comp` :

```
module IntComp : Comp =
struct
  type t = int
  let compare = compare
end

module IntSet = Set(IntComp)
```

O problema está no fato de que `IntComp` foi declarado como tendo assinatura `Comp`, e o tipo `t` em `Comp` é abstrato. Se definirmos a função `remove_dup` para remover valores duplicados usando esse módulo `IntSet`:

```
let remove_dup l =
  let conj = List.fold_left IntSet.adiciona IntSet.vazio l in
  IntSet.lista conj
```

Tudo parece igual a antes, mas essa função não funciona da forma esperada.

```
# remove_dup [1; 2; 1; 8; 7; 2];;
Error: This expression has type int but an expression
was expected of type IntComp.t
```

Esse erro é comum quando trabalhamos com funtores. O tipo do módulo `IntSet` nesse caso é interessante:

```
# module IntSet = Set(IntComp);;;;
module IntSet :
  sig
    type t = Set(IntComp).t = No of IntComp.t * t * t | Folha
    val vazio : t
    val busca : t -> IntComp.t -> t option
    val adiciona : t -> IntComp.t -> t
    val lista : t -> IntComp.t list
  end
```

Veja que o tipo dos elementos do conjunto é, na verdade, `IntComp.t`, e não `int`. Para entender o problema, vamos rever a assinatura `Comp`:

```
module type Comp =
sig
  type t

  val compare : t -> t -> int
end
```

O tipo `t` na assinatura `Comp` é *abstrato*. Isso significa que usuários de módulos com essa assinatura não podem acessar `t` diretamente. Como `IntComp` tem assinatura `Comp`, o tipo

`IntComp.t` também é abstrato. Portanto, usuários externos a esse módulo (como o módulo `IntSet`) não têm acesso a informação que `IntComp.t` é o mesmo tipo que `int`. Isso é o que vemos no erro ao tentar usar `remove_dup`.

Mas podemos manter o código dessa forma se adicionarmos essa informação: que o tipo `IntComp.t` (ou seja, o tipo `t` da assinatura `Comp`) e `int` são o mesmo. Isso é feito com uma restrição de compartilhamento (*sharing constraint*) na assinatura `Comp`, ao criar o módulo `IntComp`:

```
module IntComp : Comp with type t = int =
struct
  type t = int
  let compare = compare
end

module IntSet = Set(IntComp)
```

A assinatura do módulo `IntComp` é declarada como `Comp with type t = int`, ou seja, é a assinatura `Comp`, mas expondo o fato de que o tipo `t` da assinatura deve ser igualado ao tipo `int`. A assinatura do módulo `IntSet` criado neste exemplo mostra claramente a diferença para a versão anterior:

```
# module IntSet = Set(IntComp);;
module IntSet :
  sig
    type t = Set(IntComp).t = No of int * t * t | Folha
    val vazio : t
    val busca : t -> int -> t option
    val adiciona : t -> int -> t
    val lista : t -> int list
  end
```

Agora o tipo do elemento em `IntSet` é mesmo `int`, embora o tipo `IntComp.t` ainda exista na assinatura do módulo `IntComp`, como podemos ver no REPL:

```
# module IntComp : Comp with type t = int =
struct type t = int let compare = compare end;;
```

```
module IntComp : sig type t = int val compare : t -> t -> t end
```

A restrição de compartilhamento apenas guarda o fato de que `IntSet.t` é igual a `int`. Uma outra possibilidade é apagar completamente o tipo `IntComp.t` e substituí-lo por `int` em todos os lugares onde ele ocorre. Isso não é obrigatório, mas reduz o número de tipos na assinatura e pode resultar em código um pouco mais eficiente. Para fazer isso, usamos uma restrição com `type t := int` em vez de `type t = int`:

```
# module IntComp : Comp with type t := int =
struct type t = int let compare = compare end;;
module IntComp : sig val compare : int -> int -> int end
```

O módulo `IntComp` agora não possui mais um tipo `t`, apenas a função `compare` que funciona com valores inteiros. Infelizmente, agora `IntComp` não segue mais a assinatura `Comp` por não ter um tipo `t`:

```
# module IntSet = Set(IntComp);;
Error: Signature mismatch:
Modules do not match:
  sig val compare : int -> int -> int end
is not included in
  Comp
The type `t' is required but not provided
```

Isso é um problema, pois `IntComp` é um módulo que será usado como parâmetro para um funtor, e deve seguir a assinatura `Comp`. Para módulos que não precisam seguir uma assinatura específica, a restrição com substituição de tipo pode ser uma boa opção.

Conjuntos e mapas na biblioteca padrão

O módulo `Set` que criamos nos exemplos anteriores é similar, em termos de organização, ao módulo `Set` definido na biblioteca padrão da linguagem. Também é necessário usar um funtor para criar módulos para conjuntos com elementos de tipos específicos.

Na biblioteca padrão, esse funtor é chamado `Set.Make`. A forma de criar um módulo para conjuntos de um tipo, usando a biblioteca padrão, é:

```
module IntSet =
  Set.Make(struct type t = int let compare = compare end)
```

Isso é quase o mesmo que fizemos no exemplo anterior, mas o módulo `IntSet` criado desta forma tem várias operações além de buscar por um elemento e adicionar um elemento. Os conjuntos da biblioteca padrão também são estruturas puramente funcionais (adicionar um elemento a um conjunto cria um novo conjunto, sem alterar o original) e são implementados com árvores binárias autobalanceadas.

O módulo `Map` da biblioteca padrão funciona exatamente da mesma forma, mas para mapas que associam os elementos de um tipo de *chaves* a elementos de um outro tipo (os *valores*). Também é necessário criar módulos para mapas com tipos de chave específicos usando o funtor `Map.Make`.

8.5 EXTENSÃO DE ESTRUTURAS E ASSINATURAS

A determinação se um módulo segue ou não uma assinatura em OCaml é *estrutural*: um módulo `M` segue uma assinatura `S` se `M` tem todos os itens previstos na assinatura `S`, com os tipos corretos (para itens que forem valores). O módulo `M` não precisa ser declarado como tendo a assinatura `S`, e pode incluir outros itens que não façam parte de `S`.

Em geral, a tipagem em OCaml é estrutural, diferente da *tipagem por nome* que acontece em muitas linguagens orientadas a objeto. A tipagem estrutural tem similaridades com a ideia de *duck typing*.

Tipagem estrutural para módulos funciona bem quando queremos estender estruturas ou assinaturas para incluir novos itens. Vamos lembrar da assinatura `Comp` que especifica a interface para tipos comparáveis:

```
module type Comp =
sig
  type t

  val compare : t -> t -> int
end
```

Agora, digamos que precisamos especificar uma interface para tipos que podem ser comparados e para os quais é possível calcular um valor de *hash* para armazenar em uma tabela. Não existe herança de assinaturas, mas uma assinatura pode estender outra:

```
module type CompHash =
sig
  include Comp

  val hash : t -> int
end
```

A assinatura `CompHash` inclui tudo que existe na assinatura `Comp` (usando `include`) e adiciona a função `hash` que calcula o valor *hash* para um elemento do tipo `t`. Veja que o tipo `t` vem da assinatura `Comp`, mas todos os itens de `Comp` são incluídos diretamente na assinatura `CompHash`, sem precisar referenciar a assinatura de origem.

Como a tipagem dos módulos é estrutural, a declaração anterior é exatamente equivalente à seguinte forma:

```
module type CompHash =
sig
  type t

  val compare : t -> t -> int

  val hash : t -> int
end
```

Ou seja, o `include` é apenas um atalho para incluir todos os itens de uma assinatura em outra, mas isso não cria automaticamente nenhuma relação entre elas. Entretanto, como qualquer módulo que satisfaz a assinatura `CompHash` também satisfaz `Comp`, um módulo `M` com assinatura `CompHash` pode ser passado para um funtor que espera um módulo com assinatura `Comp` como parâmetro.

Para criar um módulo que implementa a assinatura `CompHash`, podemos também estender uma estrutura já existente. Repetimos aqui o módulo `IntComp` que implementa a assinatura `Comp`:

```
module IntComp : Comp with type t = int =
struct
  type t = int
  let compare = compare
end
```

Agora vamos criar o módulo `IntCompHash` que implementa a assinatura `CompHash`:

```
module IntCompHash : CompHash with type t = int =
struct
  include IntComp

  let hash n = n
end
```

A função `hash` é implementada de forma que o código *hash* de um inteiro é o próprio inteiro. Note que, se `IntComp` é declarado sem a restrição `with type t = int`, tentar criar `IntCompHash` pela inclusão de `IntComp` não funciona.

Incluir um módulo em outro, ou uma assinatura em outra, cria um alto acoplamento entre módulos (ou assinaturas), de modo que, se o módulo incluído muda, o módulo que o inclui muda também, e isso pode não ser desejado. Isso não é diferente do acoplamento estabelecido entre classes quando se usa herança, uma característica OO que já foi amplamente criticada pela comunidade. Como todas

as outras opções para projeto de programas, essa também pode ser bem utilizada nas situações certas, desde que se entenda os custos de seu uso.

8.6 MÓDULOS DE PRIMEIRA CLASSE

Tradicionalmente, os módulos nas linguagens da família ML são um conceito estático, presente durante a compilação como forma de separação de espaços de nomes e de abstração, mas sem um grande impacto durante a execução do programa.

Os módulos de primeira classe são uma extensão do sistema de módulos em OCaml que permite manipular módulos dinamicamente. Essa extensão vem ganhando mais força e importância a cada nova versão da linguagem, e podem se tornar ainda mais importante se a proposta de implícitos modulares for adicionada a OCaml no futuro. Módulos de primeira classe são uma característica avançada da linguagem, e aqui teremos uma introdução a eles.

Um caso de uso para módulos de primeira classe é a reconfiguração dinâmica de um programa. Como exemplo, podemos imaginar um sistema de extensões ou *plugins*: cada *plugin* seria um módulo que satisfaz a mesma assinatura, e os *plugins* utilizados poderiam mudar durante a execução do programa, sem precisar recompilar e sem que o programa principal tenha conhecimento de cada um (para isso, também é necessário usar o suporte dos compiladores OCaml para link dinâmico).

Outra possibilidade é escolher, em tempo de execução, uma entre várias implementações de uma mesma assinatura. Podemos guardar todos os módulos que implementam a assinatura em uma lista ou tabela *hash* e, em tempo de execução (talvez de acordo com uma escolha do usuário), decidir qual deles usar.

Para tratar um módulo como um valor de primeira classe da linguagem, basta usar uma expressão `module M : T`, em que `M` é uma expressão que designa um módulo e `T` é uma assinatura, que é opcional se a assinatura puder ser inferida automaticamente. Na maioria dos contextos, é necessário usar parênteses em torno da expressão de módulo. Usando o módulo `IntComp` e a assinatura `Comp` de exemplos anteriores neste capítulo:

```
# (module IntComp : Comp);;
- : (module Comp) = <module>
```

O REPL mostra que esse é um valor de módulo (com o indicador `<module>`, similar ao usado para funções) e que seu tipo é `module Comp`. Essa expressão empacota um módulo como um valor da linguagem. Uma vez transformado em um valor, ele pode ser passado como parâmetro para funções, armazenado em variáveis e estruturas etc.:

```
# let m = (module IntComp : Comp);;
val m : (module Comp) = <module>

# type comparador = { nome: string; modulo : (module Comp) };;
type comparador = { nome : bytes; modulo : (module Comp); }

# let comp1 = { nome = "comparador de inteiros"; modulo = m };;
val comp1 : comparador = {nome = "comparador de inteiros";
                          modulo = <module>}
```

Funções que recebem um módulo de primeira classe como parâmetro precisam desempacotar o módulo que está no valor usando uma expressão `val m : T`, em que `m` é um módulo de primeira classe e `T` é a sua assinatura:

```
module type Mensagem = sig
  val msg : string
end

module Hello : Mensagem = struct
  let msg = "Hello"
end
```

```
module QuePasa : Mensagem = struct
  let msg = "Que pasa?"
end

let imprime_msg m =
  let module M = (val m : Mensagem) in
  print_endline M.msg

let mensagens : (module Mensagem) list =
  [(module Hello : Mensagem); (module QuePasa : Mensagem)]
```

Com a lista `mensagens`, podemos imprimir todas as mensagens da lista:

```
# List.iter imprime_msg mensagens;;
Hello
Que pasa?
- : unit = ()
```

Obviamente, esse exemplo é muito simples e não seria necessário usar módulos de primeira classe para variar uma mensagem. Mas o conteúdo dos módulos poderia ser qualquer coisa, e esse exemplo demonstra como tratá-los como valores que podem ser manipulados da mesma forma que quaisquer outros valores da linguagem.

Os módulos de primeira classe podem ser usados para criar módulos parametrizados, substituindo os funtores. Mas em geral é melhor usar os funtores, a não ser que seja necessário mudar a configuração dos módulos dinamicamente.

Como podemos ver nos exemplos, o uso de módulos de primeira classe precisa de mais anotações explícitas e torna a notação mais pesada nos programas. Além disso, as operações necessárias para empacotar e desempacotar módulos como valores impedem algumas otimizações e podem resultar em código menos eficiente.

Todas essas questões devem ser pesadas na hora de usar módulos de primeira classe. Entretanto, existem situações em que

eles são a única opção (reconfiguração dinâmica do programa, por exemplo).

Como já foi comentado antes, os módulos e todas as suas possibilidades vistas neste capítulo são uma parte complexa da linguagem OCaml. Isso afeta principalmente os iniciantes, já que muitas dessas características existem primariamente para organizar programas de maior porte.

Um novato na linguagem não deve se preocupar demais em entender tudo sobre módulos inicialmente, já que muita coisa se torna mais clara à medida que se ganha mais experiência lendo e escrevendo código em OCaml.

Capítulo 9

EXEMPLO: ÁRVORES DE DECISÃO

Neste capítulo, mais uma vez analisaremos um exemplo de programa completo em vez de apenas trechos de código. O programa estudado usa conceitos de *aprendizagem de máquina* para construir uma árvore de decisão para classificar informações.

Para exemplificar o uso das árvores de decisão, vamos usar como dados o problema do *Titanic*, encontrado no Kaggle (no endereço https://www.kaggle.com/c/titanic-gettingStarted). Kaggle é um site que hospeda competições de análise de dados nas quais qualquer um pode participar. Antes, examinaremos alguns conceitos relacionados ao aprendizado de máquina, que é uma área da computação cujas técnicas usaremos neste capítulo.

9.1 O PROBLEMA DO TITANIC

O problema do Titanic consiste em analisar os dados dos passageiros do navio para tentar prever a sobrevivência de cada um após o naufrágio. Obviamente já sabemos quais passageiros sobreviveram e quais não, mas criar um programa que consiga prever isso corretamente, baseado nos dados dos passageiros, pode elucidar os motivos que levaram à sobrevivência ou não de cada um.

Vamos começar, como é comum em projetos que envolvem a análise de dados, cuidando da obtenção dos dados, e explorando as

informações neles para tirar algumas primeiras conclusões. Algumas das análises que vamos fazer são similares ao tutorial presente no site da Kaggle, em https://www.kaggle.com/c/titanic-gettingStarted/details/getting-started-with-python.

Os dados disponibilizados pelo Kaggle estão em dois arquivos, um de *treinamento* e um de *teste*. Esses nomes são convencionais nas áreas de aprendizado de máquina e mineração de dados, como veremos posteriormente. Basicamente, o arquivo de treinamento contém dados completos que servem para que tentemos identificar os padrões que ligam os dados com o resultado final; no caso, que dados determinam a sobrevivência de um passageiro do Titanic.

Inicialmente vamos tentar identificar correlações entre os dados manualmente, através da exploração das informações. Depois veremos como usar técnicas do aprendizado de máquina para encontrar essas relações automaticamente. O arquivo de testes contém dados diferentes do de treinamento, e é usado para verificar o que foi treinado, ou seja, para ver se as correlações entre os dados são realmente gerais e não só uma característica dos dados de treinamento.

No Kaggle, os dados de teste são usados para avaliar a solução do usuário e comparar com as soluções de outros. Por isso o arquivo de teste não contém a informação sobre a sobrevivência dos passageiros.

Começamos discutindo sobre como os dados estão organizados e como ler as informações em um programa OCaml.

Obtenção dos dados

No aprendizado de máquina e áreas correlatas, é comum usar os dados em um formato tabular: eles são divididos em um conjunto de itens, e cada item possui um conjunto de atributos que o

identificam. No problema do Titanic, cada item de dados é relativo a uma pessoa que viajou no navio, e os atributos descrevem dados do passageiro que podem ser relevantes para a tarefa, como idade e em que classe viajava na embarcação.

Os dados estão disponíveis no formato CSV (*Comma-Separated Values*), um formato comum para dados em formato texto que contém um item em cada linha, e os atributos de cada item estão separados por vírgulas. O arquivo de treinamento se chama `train.csv` e o de teste se chama `test.csv`. Ambos os arquivos podem ser obtidos no site da competição ou no código de exemplo do livro.

Para ter uma ideia da organização dos dados, segue uma versão resumida de algumas linhas no começo do arquivo `train.csv`:

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,...
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
5,0,3,"Allen, Mr. William Henry",male,35,0,0,373450,8.05,,S
6,0,3,"Moran, Mr. James",male,,0,0,330877,8.4583,,Q
7,0,1,"McCarthy, Mr. Timothy J",male,54,0,0,17463,51.8625,E46,S
8,0,3,"Palsson, Master. Gosta Leonard",male,2,3,1,349909,
      21.075,,S
14,0,3,"Andersson, Mr. Anders Johan",male,39,1,5,347082,
      31.275,,S
```

A primeira linha é um cabeçalho, contendo o nome de cada atributo dos itens, na ordem em que aparecem nas linhas seguintes. Para entender os dados, é preciso saber o que significa e como é representado cada atributo. Geralmente, essa informação está presente em alguma descrição dos dados, e é imprescindível para fazer uma análise.

Para os dados do Titanic, os atributos são:

- `PassengerId` é um identificador único para cada passageiro. Esse identificador foi determinado pelos criadores do conjunto de dados, e não deve ser usado

para previsões.

- `Survived` indica se o passageiro sobreviveu ou não, com o valor 1 para o caso de ter sobrevivido, e 0 para caso de morte.
- `Pclass` é a classe do passageiro no navio: 1 para primeira classe, 2 para segunda e 3 para terceira. Esse atributo é usado como indicador para a classe social do passageiro.
- `Name` é o nome do passageiro, no formato *sobrenome, nome*.
- `Sex` é o gênero do passageiro: `male` para homem, `female` para mulher.
- `Age` é a idade, em anos, podendo incluir frações. Idades estimadas são indicadas por um número xx.5.
- `SibSp` é o número de irmãos, irmãs e cônjuges a bordo (*siblings and spouses*).
- `Parch` é o número de pais e filhos do passageiro a bordo (*parents and children*).
- `Ticket` é uma string com o identificador da passagem.
- `Fare` indica o preço pago pela passagem.
- `Cabin` é uma string que identifica a cabine onde o passageiro viajou.
- `Embarked` indica o porto de embarque no navio, sendo C para Cherbourg, Q para Queenstown, e S para Southampton.

Mais alguns detalhes sobre esses dados (por exemplo, o que exatamente conta como *irmão ou cônjuge*) estão disponíveis na página da competição (em https://www.kaggle.com/c/titanic-gettingStarted/data).

Olhando para a segunda linha da amostra do arquivo de treinamento:

```
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
```

Descobrimos que este passageiro, chamado *Mr. Owen Harris Braund*, recebeu o identificador 1, não sobreviveu, viajou na terceira classe, era um homem, tinha 22 anos de idade e embarcou em Southampton, entre outras informações. A informação sobre a cabine onde o passageiro viajou não está presente, como se nota pelas duas vírgulas perto do final da linha.

Em conjuntos de dados reais, é comum ter dados faltantes por vários motivos. Ao fazer uma análise de dados, é importante levar essa possibilidade em consideração.

Para ler os dados em programas OCaml, vamos usar a biblioteca `ocaml-csv`, que dá suporte à leitura de arquivos CSV. Para quem usa OPAM, a instalação dessa biblioteca é feita com `opam install csv`. Para um REPL configurado para usar a `findlib` (vide instruções de instalação no capítulo *Introdução* ou no site do livro), pode-se usar a biblioteca com a diretiva `#require`:

```
# #require "csv";;
/Users/andrei/.opam/4.02.0/lib/csv: added to search path
/Users/andrei/.opam/4.02.0/lib/csv/csv.cma: loaded
# Csv.load;;
- : ?separator:char -> ?excel_tricks:bool ->
    string -> Csv.t = <fun>
```

O nome da biblioteca na `findlib` é `csv`, e o módulo que contém a interface da biblioteca se chama `Csv`. A função `Csv.load` carrega os dados a partir de um arquivo CSV e é a que usaremos aqui (o tipo da função `Csv.load` inclui argumentos *opcionais*, como veremos no capítulo *Parâmetros rotulados*). Basta passar o nome do arquivo CSV como parâmetro:

```
# let csv = Csv.load "train.csv";;
val csv : Csv.t =
  [["PassengerId"; "Survived"; "Pclass"; "Name"; "Sex"; "Age";
    "SibSp"; "Parch"; "Ticket"; "Fare"; "Cabin"; "Embarked"];
   ["1"; "0"; "3"; "Braund, Mr. Owen Harris"; "male"; "22";
    "1"; "0"; "A/5 21171"; "7.25"; ""; "S"];
```

...

O tipo de retorno de `Csv.load` é um valor `Csv.t`, que é um sinônimo para `string list list`, ou seja, uma lista de listas de strings. Cada linha do arquivo CSV é representada por uma lista de strings, e todas elas são reunidas em uma lista de listas. Na saída da função `Csv.load`, podemos ver a primeira linha, que é o cabeçalho, e a segunda, que são os dados do passageiro com identificador 1.

A representação com strings é simples porque vem diretamente da leitura do arquivo CSV original, mas para tratar e analisar os dados é mais interessante trabalhar com tipos mais interessantes. Por exemplo, o atributo `Survived` pode ser um booleano indicando a sobrevivência, e o gênero pode ser indicado com um tipo variante com duas possibilidades. Essa tradução para uma representação mais rica dos dados é mostrada a seguir.

Representação tipada dos dados

Com base na descrição de cada atributo do conjunto de dados, definimos um tipo de registro para guardar as informações de cada passageiro. Para atributos numéricos, usaremos os tipos `int` e `float`, mas para os outros podemos definir tipos variantes específicos:

```
type classe = Primeira | Segunda | Terceira
type genero = Masc | Fem
type porto = Cherbourg | Queenstown | Southampton
```

Esses tipos e seus valores são autoexplicativos. Para o atributo de sobrevivência, usamos o tipo `bool`. Uma questão nesse ponto é saber que atributos podem estar faltando no arquivo de dados, e para isso precisamos fazer alguma exploração dos arquivos disponíveis (de treinamento e de teste). Outra opção seria assumir que todos os dados podem estar faltando e usar `option` em tudo.

Isso é possível, porém complicaria um pouco a análise. Vamos optar por analisar, nos arquivos de dados, que atributos podem faltar e determinar os tipos a partir disso.

Considerando os dados retornados por `Csv.load` como uma tabela, a função `dados_em_falta` constrói uma lista contendo as coordenadas (linha e coluna) nas quais não há informação, e a função `colunas_em_falta` usa isso para determinar quais colunas estão faltando no conjunto de dados completo:

```
let dados_em_falta csv =
  let coluna i j el = if el = "" then [(i, j)] else [] in
  let linha i lin =
    lin |> List.mapi (coluna i) |> List.concat
  in
  csv |> List.mapi linha |> List.concat

let colunas_em_falta csv =
  let faltas = dados_em_falta csv in
  let module IntSet =
    Set.Make (struct type t = int let compare = compare end)
  in
  let set_lista s = IntSet.fold (fun el lis -> el :: lis) s [] in
  let adiciona_col set (lin, col) = IntSet.add col set in
  faltas
  |> List.fold_left adiciona_col IntSet.empty
  |> set_lista
```

A função `colunas_em_falta` usa o funtor `Set.Make` para criar localmente um módulo com operações para conjuntos que armazenam inteiros, usando a forma `let module`. Para cada item na lista produzida por `dados_em_falta`, a função `colunas_em_falta` adiciona apenas o número da coluna faltante em um conjunto, o que evita duplicações. Depois, é feito um `fold` no conjunto para coletar todos números de colunas vistos e produzir a lista de saída.

O operador `|>` é muito usado nas funções deste capítulo. Lembre-se de que esse operador apenas inverte a ordem na aplicação de função: `x |> f` é o mesmo que `f x`. O operador

também é chamado de *pipeline* e serve para deixar mais claro o processo pelo qual os dados passam: por exemplo, no final da função `colunas_em_falta`, começamos com os dados na variável `faltas` (que é uma lista de pares indicando as coordenadas dos valores faltantes), passamos essa lista para um `fold_left`, que cria um conjunto com as colunas faltantes, e depois esse resultado é passado para a função local `set_lista` que transforma o conjunto em uma lista. Se usássemos a ordem de aplicação normal da linguagem, ou o operador `@@`, precisaríamos ler esse processamento dos dados ao contrário, da direita para a esquerda.

Usando a função `colunas_em_falta` nos dados de treinamento carregados na variável `csv`, obtemos:

```
# colunas_em_falta csv;;
- : int list = [11; 10; 5]
```

A ordem das colunas é a vista no arquivo CSV, contando a partir do zero. O resultado mostrado indica que só três atributos podem estar faltando: a coluna 5 equivale ao atributo `idade`, a coluna 10 equivale ao atributo `cabine` e a coluna 11 representa o atributo porto de `embarque`.

Isso é para o arquivo de treinamento. Fazendo a mesma análise para o arquivo de teste, obtemos o seguinte resultado:

```
# let csv_test = Csv.load "test.csv";;
val csv_test : Csv.t =
  [["PassengerId"; "Pclass"; "Name"; "Sex"; "Age"; "SibSp";
    "Parch"; "Ticket"; "Fare"; "Cabin"; "Embarked"];
   ["892"; "3"; "Kelly, Mr. James"; "male"; "34.5"; "0"; "0";
    "330911"; "7.8292"; ""; "Q"];
    ...

# colunas_em_falta csv_test;;
- : int list = [9; 8; 4]
```

Os atributos faltantes no arquivo de teste estão nas colunas 4, 8 e 9. Mas é importante notar, como vemos no valor mostrado para a

variável `csv_test`, que o arquivo de teste não inclui o atributo `Survived`, o que faz sentido pois esse é o resultado que precisamos calcular para o arquivo de testes, e ele só é conhecido pelo Kaggle e avaliado quando submetemos uma solução. Portanto, as colunas desse resultado correspondem às colunas 5, 9 e 10 do arquivo de treinamento.

As colunas 5 e 10 também possuem atributos em falta no arquivo de treinamento, logo, não apresentam novidade. Mas o arquivo de testes tem atributos faltantes na coluna 9 também, que equivale ao atributo `preco` da passagem.

Resumindo o resultado dos dois conjuntos de dados, os atributos que podem estar em falta são: idade, cabine, porto de embarque e preço. Agora podemos definir o tipo de registro que representa todas as informações de um passageiro:

```
type classe = Primeira | Segunda | Terceira
type genero = Masc | Fem
type porto = Cherbourg | Queenstown | Southampton

type passageiro = {
  id         : int;
  sobreviveu : bool;
  classe     : classe;
  nome       : string;
  gen        : genero option;
  idade      : float;
  irmpar     : int;
  paisfilhos : int;
  passagem   : string;
  preco      : float;
  cabine     : string;
  embarque   : porto option
}
```

Vamos representar um dado em falta para os atributos `idade` e `preco` com o valor `0.0`, supondo que esse valor não ocorre normalmente. Como a cabine é representada por uma string, um valor faltante é apenas uma string vazia e não deve afetar nossa

análise.

O porto de embarque é representado com o tipo `porto option`, pois o valor do atributo pode estar em falta. Além disso, representamos o atributo `gen` com o tipo `genero option`. Como o gênero é representado com strings, o uso de um tipo `option` é feito considerando a possibilidade de valores mal formados, que não seguem o formato esperado.

Com os tipos definidos, criamos algumas funções para converter os valores do arquivo, em formato string, para valores dos tipos definidos. Apesar de já existirem as funções `int_of_string` e `float_of_string` na biblioteca padrão para converter de string para `int` ou `float`, elas disparam uma exceção se a string de entrada for mal formada (por exemplo, uma string vazia). Por isso, definimos funções `ler_int` e `ler_float` para tratar dos casos em que o valor não existe ou é mal formado.

As funções de leitura são bastante simples:

```
let ler_sobr s = if s = "0" then false else true

let ler_classe s =
  match s with
  | "1" -> Primeira
  | "2" -> Segunda
  | "3" -> Terceira
  | _ -> failwith "Classe inexistente"

let ler_genero s =
  match s with
  | "male" -> Some Masc
  | "female" -> Some Fem
  | _ -> None

let ler_int s =
  try
    int_of_string s
  with
    Failure _ -> 0

let ler_float s =
```

```
  try
    float_of_string s
  with
    Failure _ -> 0.0

let ler_embarq s =
  match s with
  | "C" -> Some Cherbourg
  | "Q" -> Some Queenstown
  | "S" -> Some Southampton
  | _ -> None
```

Juntamos as funções de leitura em uma função usada para converter os dados brutos em dados tipados. Como o arquivo de testes não contém a coluna com o atributo `Survived`, precisamos de duas funções muito parecidas para ler do arquivo de treinamento e de testes.

Mostramos a seguir a que lê os dados de teste. A função para dados de treinamento é muito similar e pode ser vista no código-fonte completo.

```
let ler_dados_treino nome =
  let csv = Csv.load nome in
  let ler_pass [id; sobr; cls; nome; gen; idade; ip; pf; pass;
                prec; cab; emb] =
    { id = ler_int id;
      sobreviveu = ler_sobr sobr;
      classe = ler_classe cls;
      nome = nome;
      gen = ler_genero gen;
      idade = ler_float idade;
      irmpar = ler_int ip;
      paisfilhos = ler_int pf;
      passagem = pass;
      preco = ler_float prec;
      cabine = cab;
      embarque = ler_embarq emb }
  in
  List.map ler_pass @@ List.tl csv
```

Essas funções usam um padrão refutável e, por isso, o compilador mostra um aviso quando elas são processadas. Entretanto, conhecemos o formato do arquivo e sabemos que esse

padrão não deve falhar. Mesmo assim, se tentarmos ler um arquivo CSV com formato diferente usando essa função, uma exceção será disparada e o processo será terminado. Um programa mais robusto poderia testar se o formato do arquivo de dados está de acordo com o esperado, ou pelo menos tratar a exceção `Match_failure` .

Com as funções de leitura, obtemos os dados em uma forma tipada, que é muito mais fácil de manipular. Os primeiros dois itens lidos são mostrados a seguir.

```
# let dados = ler_dados_treino "train.csv";;
val dados : passageiro list =
  [{id = 1; sobreviveu = false; classe = Terceira;
    nome = "Braund, Mr. Owen Harris"; gen = Some Masc;
    idade = 22.; irmpar = 1; paisfilhos = 0;
    passagem = "A/5 21171"; preco = 7.25;
    cabine = ""; embarque = Some Southampton};
   {id = 2; sobreviveu = true; classe = Primeira;
    nome = "Cumings, Mrs. John Bradley (Florence Briggs Thayer)";
    gen = Some Fem; idade = 38.; irmpar = 1; paisfilhos = 0;
    passagem = "PC 17599"; preco = 71.2833; cabine = "C85";
    embarque = Some Cherbourg};
  ...
```

Mulheres e crianças primeiro

Uma boa forma de começar a análise dos dados é calcular a sobrevivência de homens e de mulheres. Para isso, precisamos filtrar os dados para selecionar itens de acordo com o gênero, e reduzir os dados filtrados para resumir a taxa de sobrevivência de cada gênero.

Em algumas linguagens, isso é difícil de fazer ou requer o uso de bibliotecas especiais para lidar com dados, mas em OCaml é apenas uma aplicação comum das funções de alta ordem da programação funcional. Definimos predicados para testar se um passageiro é homem ou mulher, e as funções `apenas_homens` e `apenas_mulheres` usando `filter` :

```
let homem p = p.gen <> Some Fem
let mulher p = p.gen = Some Fem
```

```
let apenas_homens d =
  List.filter homem d

let apenas_mulheres d =
  List.filter mulher d
```

Definimos uma função geral que calcula a taxa de sobrevivência de qualquer subconjunto de passageiros que satisfaz um predicado:

```
let taxa_sobrev_pred pred d =
  let pass = List.filter pred d in
  let sobr =
    pass |> List.fold_left (fun s p ->
                            if p.sobreviveu then s + 1
                            else s) 0
  in
  (float sobr) /. (float @@ List.length pass)
```

Com essa função, podemos calcular a taxa de sobrevivência geral:

```
# taxa_sobrev_pred (fun p -> true) dados;;
- : float = 0.383838383838383812
```

Ou seja, dos passageiros do conjunto de treinamento, apenas um pouco mais de 38% sobreviveram. Também é fácil calcular a taxa de sobrevivência de homens e mulheres, separadamente, usando os predicados definidos antes:

```
# taxa_sobrev_pred homem dados;;
- : float = 0.188908145580589243

# taxa_sobrev_pred mulher dados;;
- : float = 0.7420382165605095
```

Enquanto apenas 18.9% dos passageiros homens sobreviveram, quase três quartos (74.2%) das mulheres saíram vivas da viagem. Isso nos dá uma ideia para um classificador simples, criado manualmente: para qualquer passageiro mulher, o classificador prevê sua sobrevivência; se for homem, o classificador prevê a não sobrevivência.

Para submeter uma resposta do problema ao Kaggle, é preciso calcular a sobrevivência de cada passageiro do arquivo de teste, e escrever um arquivo CSV na saída contendo pelo menos o identificador do passageiro e o atributo `Survived` com valor 1 para sobrevivência, e 0 para não sobrevivência. O classificador que decide a sobrevivência por gênero pode ser escrito com pouco código:

```
let sobrevivencia_por_genero d =
  d |> List.map (fun p ->
                 if mulher p then (p.id, 1)
                 else (p.id, 0))
```

Isso é o suficiente para calcular a resposta. Escrevemos também uma função para escrever os dados, e uma outra função que lê todos os dados de teste e escreve os resultados no arquivo `genero.csv`:

```
let escreve_resultado res nome =
  let arqout = open_out nome in
  Printf.fprintf arqout "PassengerId,Survived\n";
  List.iter (fun (id, s) ->
             Printf.fprintf arqout "%d,%d\n" id s)
            res;
  close_out arqout

let classifica_teste_por_genero arqteste arqsaida =
  let dados_teste = ler_dados_teste arqteste in
  let resultado = sobrevivencia_por_genero dados_teste in
  escreve_resultado resultado arqsaida
```

Uma chamada à função `classifica_teste_por_genero` lê o arquivo de teste, chama o classificador por gênero e escreve o resultado. Podemos gerar o arquivo abrindo uma nova sessão do REPL no diretório no qual se encontra o arquivo `test.csv`:

```
# #require "csv";;
/Users/andrei/.opam/4.02.0/lib/csv: added to search path
/Users/andrei/.opam/4.02.0/lib/csv/csv.cma: loaded

# #use "titanic.ml";;
type classe = Primeira | Segunda | Terceira
type genero = Masc | Fem
type porto = Cherbourg | Queenstown | Southampton
```

```
...

# classifica_teste_por_genero "train.csv" "genero.csv";;
- : unit = ()
```

Após executar esses comandos (ou o executável `genero` gerado pela compilação do projeto), o arquivo `genero.csv` deve ter sido criado no diretório atual. O arquivo gerado pode ser submetido diretamente ao Kaggle. Fazendo isso, obtemos uma pontuação de classificação de 0.76555. A figura seguinte mostra o relatório de pontuação mostrado pelo Kaggle para essa submissão.

Figura 9.1: Pontuação do classificador baseado em gênero

Esse é um resultado razoável. Para essa competição, a pontuação é a porcentagem de passageiros corretamente classificados, ou seja, com o classificador por gênero conseguimos prever corretamente a sobrevivência de 76.5% dos passageiros.

Mas isso pode ser melhorado, se explorarmos mais o conjunto de dados. Alguns atributos talvez não tenham uma influência grande para a sobrevivência dos passageiros, por exemplo, o porto de embarque. Mas não é possível afirmar que algum atributo não tem influência sem analisar os dados antes. Existem outros atributos que parecem ter mais chance de influenciar o resultado, como a classe em que o passageiro viajou ou o preço pago pela passagem.

Poderíamos continuar a exploração dos dados e encontrar formas de usar a classe do passageiro e o preço para ajudar a prever a sobrevivência. Ao fazer isso, teríamos um conjunto de regras como *se o passageiro for mulher, ela sobrevive; caso contrário, se for*

um homem da primeira classe, ele sobrevive, mas se for um homem da terceira classe, não sobrevive, e assim por diante.

Em vez de analisar os dados e encontrar relações entre eles, construindo regras de classificação manualmente, vamos usar técnicas de aprendizado de máquina para tentar inferir automaticamente essas regras. As regras de classificação serão representadas em uma árvore de decisão.

9.2 UM POUCO SOBRE APRENDIZADO DE MÁQUINA

Aprendizado de máquina é uma área da computação (às vezes, considerada uma subárea da Inteligência Artificial) relacionada à criação de agentes que conseguem *melhorar* seu desempenho em alguma tarefa, ou seja, conseguem aprender com a experiência.

A área de aprendizado de máquina se originou na computação, usando técnicas principalmente da lógica matemática. Com o tempo, foi crescendo o uso de técnicas probabilísticas e estatísticas, e hoje em dia está muito relacionado com a análise de dados e com a área de estudo da Estatística. Algumas técnicas foram desenvolvidas de maneira paralela nos dois lados (computação e estatística), que hoje em dia tendem a uma unificação. As árvores de decisão foram uma das técnicas que surgiram tanto no lado da computação (com algoritmos como ID3 e C4.5) quanto no lado da estatística (por exemplo, o algoritmo CART).

Aprendizado supervisionado e não-supervisionado

As técnicas de aprendizado são normalmente divididas em duas grandes classes: as que realizam aprendizado *supervisionado* e as que fazem aprendizado *não-supervisionado*.

No aprendizado supervisionado, existe (conceitualmente) um *supervisor* que fornece ao agente exemplos que são instâncias da tarefa em questão, e mede o desempenho do agente nesta tarefa. No aprendizado não-supervisionado, o agente não tem um supervisor ou referência, e deve encontrar relações entre os dados de maneira autônoma.

A figura adiante mostra uma visão esquemática de como funciona o treinamento no aprendizado supervisionado. O agente recebe uma entrada e determina um resultado. O supervisor tem acesso a um resultado de referência, ou seja, qual deveria ser o resultado para cada entrada.

Comparando o resultado produzido pelo agente com o valor de referência, o supervisor pode fornecer um *feedback* para o agente, o que podemos chamar de *erro*: uma medida de quanto o resultado do agente está distante do que deveria ser. Com isso, o agente ajusta seu comportamento para reduzir esse erro. A ideia é de que, eventualmente, o agente saiba como produzir o resultado correto.

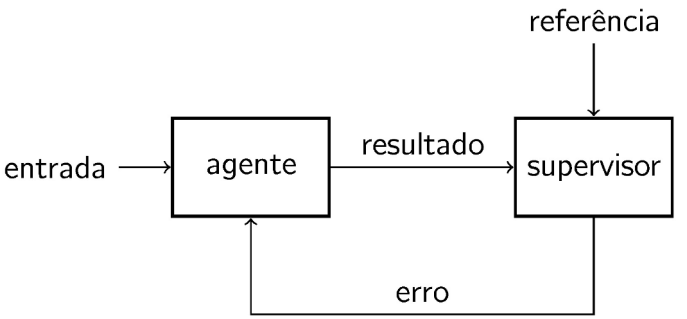

Figura 9.2: Treinamento no aprendizado supervisionado

A tarefa na qual estamos interessados é conhecida como *classificação*, e consiste em determinar a qual classe um item de dados pertence. Para o problema do Titanic, a classificação é para determinar a qual das duas classes, sobrevivente ou não-

sobrevivente, cada item de dados (pessoa) pertence. Um agente que aprenda a classificar itens de dados neste problema pode prever a sobrevivência ou não-sobrevivência mesmo de passageiros para os quais o resultado não é conhecido (como ocorre no arquivo de teste).

A divisão dos dados em um conjunto de treinamento e um conjunto de testes é comum em tarefas de classificação. A ideia é usar o conjunto de treinamento para fazer o classificador aprender a classificar os dados, e depois usar o conjunto de testes para verificar quão bem o classificador funciona para dados que não foram usados para treiná-lo.

Árvores de decisão

As árvores de decisão podem ser encaradas como estruturas de dados que representam um conjunto de regras de classificação para os itens de um conjunto de dados. Ao classificar um item, cada nó interno da árvore é associado a um teste desse item.

Começando pela raiz, realiza-se o teste associado ao nó atual e, dependendo do resultado, o processo continua por um dos filhos do nó atual. Isso continua até chegar a uma folha da árvore, que representa uma classe, que é a classe prevista para o item em questão.

Por exemplo, uma árvore de decisão equivalente à regra de classificação por gênero vai ter apenas um nó interno, que é a própria raiz, e o teste desse nó é ver se o gênero do passageiro é feminino ou masculino. Como são duas possibilidades de resultado no teste, a raiz da árvore terá dois filhos, um para cada resultado. Se o gênero do passageiro é feminino, a árvore continua em uma folha que prevê a sobrevivência; caso contrário, o nó seguinte também é uma folha, mas prevendo a não-sobrevivência. Essa árvore de decisão simples é mostrada na figura a seguir.

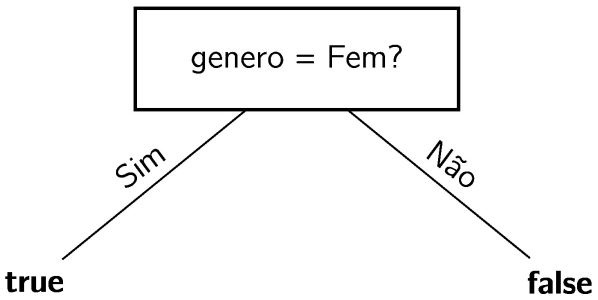

Figura 9.3: Árvore de decisão que classifica passageiros por gênero

Representação em OCaml

Uma linguagem funcional como OCaml oferece várias formas de representar as árvores de decisão. Cada nó precisa guardar um teste de um atributo e o conjunto de nós filhos. Uma representação muito usada para árvores em que cada nó pode ter um número arbitrário de filhos é usar uma lista para os filhos do nó.

Para o teste, podemos usar uma função que recebe um passageiro e retorna o número do filho selecionado na árvore. Para completar as informações de um teste, vamos usar um registro que também contém um nome para o teste e o número de valores que ele especifica. Dessa forma, o tipo para árvores de decisão seria:

```
type teste = {
  nome : string;
  f : passageiro -> int;
  nvals : int
}

type arvdec = Teste of teste * arvdec list | Result of bool
```

A função de teste retorna um inteiro para cada passageiro de entrada; um teste é composto por uma função de teste, um nome e um inteiro que especifica o número de valores retornados pela função. Uma árvore de decisão tem dois tipos de nós: nós internos são testes, com uma função de teste e uma lista de filhos. As folhas

são resultados, indicando qual das classes deve ser selecionada para o passageiro que chega à folha.

A ideia é de que a função de teste recebe o passageiro `p` que está sendo classificado pela árvore e retorna o número do filho do nó atual que deve ser seguido por `p` na árvore. Se a função de teste retorna 3, isso significa que a árvore deve seguir pelo quarto filho (a numeração começa do zero) do nó de teste, ou seja, o quarto elemento da lista de filhos.

Essa representação funciona, mas ela fere um critério importante, que é o de usar os tipos para fazer estados ilegais impossíveis de representar. Nessa representação, a função de teste deve retornar um número que é um índice válido na lista de filhos de nó, mas o tipo da árvore não impede que um teste retorne um índice inválido. Por exemplo, será fácil criar a seguinte árvore inválida:

```
let a = Teste ({ nome = "erro";
                 f = fun p ->
                     if p.gen = Some Fem then 1 else 1000;
                 nvals = 2 },
               [Resultado true; Resultado false])
```

Apesar desse problema, neste capítulo vamos usar essa representação, pois ela funciona melhor para o aprendizado de árvores do que outras alternativas simples (que não usam características mais avançadas da linguagem). Mas é importante lembrar de que isso pode não ser a ideal para um programa mais robusto que use árvores de decisão. Também é possível criar uma função de validação que verifica se uma árvore contém algum teste inválido.

Uma representação alternativa seria definir os testes de maneira que eles retornem diretamente as subárvores que são filhas do nó de teste:

```
type teste_1 = passageiro -> arvdec_1
and arvdec_1 = Teste of teste_1 | Result of bool
```

Nessa representação, um teste retorna uma árvore de decisão para cada passageiro testado, e uma árvore mais uma vez contém dois tipos de nós: nós internos são testes, e folhas são resultados. Como existe uma dependência mútua entre os tipos `teste` e `arvdec`, eles têm de ser declarados juntamente com `type .. and`. O problema é que ela dificultaria a construção das árvores de decisão automaticamente através dos dados, que veremos mais para a frente.

Vamos construir a árvore de decisão que classifica passageiros por gênero (a árvore da figura anterior) usando a representação escolhida:

```
let test_f_genero p =
  if p.gen = Some Fem then 0 else 1

let teste_genero = {
  nome = "genero";
  f = test_f_genero;
  nvals = 2;
}

let arv_genero =
  Teste (teste_genero, [Result true; Result false])
```

Essa representação para árvores de decisão não é tão compacta quanto seria o ideal, porque é preciso escrever o código das funções de teste, que em geral apenas testam o valor de um atributo. Isso poderia ser melhorado usando um pouco de metaprogramação (o que não faremos aqui). De qualquer forma, dada uma árvore, a função que classifica um passageiro é simples:

```
let rec aplica_arvore arv p =
  match arv with
  | Result true -> (p.id, 1)
  | Result false -> (p.id, 0)
  | Teste (t, l) -> aplica_arvore (List.nth l (t.f p)) p
```

Essa função já retorna um resultado no formato que precisamos para escrever o CSV para o Kaggle: um par com o identificador do passageiro e `1` para sobrevivência, `0` para não sobrevivência. Aplicar essa função a um grupo de passageiros é apenas um `map`, e podemos usar a função `escreve_resultado` sem alterações. Com essas peças, podemos escrever a função `classifica_arv_genero`:

```
let aplica_arvore_dados arv d =
  List.map (aplica_arvore arv) d

let classifica_arv_genero arqteste arqsaida =
  let dados_teste = ler_dados_teste arqteste in
  let resultado = aplica_arvore_dados arv_genero dados_teste in
  escreve_resultado resultado arqsaida
```

A função `classifica_arv_genero` gera um arquivo CSV exatamente idêntico ao arquivo gerado por `classifica_teste_por_genero`. Entretanto, nesse caso podemos mudar a árvore de decisão usada mais facilmente.

Usamos uma árvore predefinida, baseada na regra de classificação encontrada pela exploração dos dados. As árvores de decisão ficam mais interessantes quando usamos um processo de treinamento supervisionado para inferir uma árvore de decisão automaticamente a partir dos dados. Como vantagem adicional, a árvore inferida no processo de treinamento pode ser interpretada para revelar que relações entre os dados foram encontradas.

9.3 INFERÊNCIA DE ÁRVORES COM ID3

Existem vários algoritmos para inferir uma árvore de decisão automaticamente a partir de um conjunto de dados de treinamento. Neste capítulo, vamos detalhar e implementar um dos algoritmos mais simples, conhecido como ID3. Esse algoritmo é simples de entender e implementar, mas possui limitações importantes que muitas vezes o tornam inadequado para uso em situações reais.

Algoritmos como C4.5 e C5.0 (e variantes deles) são sucessores do ID3 que superam as limitações e são mais usados na prática.

Uma árvore de decisão é composta por testes que determinam o caminho de um item de dados da raiz da árvore até uma folha. Inferir uma árvore a partir dos dados é selecionar quais testes serão feitos e como eles serão organizados na estrutura da árvore. Um algoritmo simples como ID3 usa apenas testes *simples*, testes que verificam apenas o valor de apenas um atributo de cada vez.

Além disso, o algoritmo ID3 tem suporte apenas a atributos *categóricos*, cujo valor é uma entre um conjunto discreto (e geralmente pequeno) de categorias. No problema do Titanic, atributos como `classe` e `porto` são categóricos, mas atributos numéricos como `idade` e `preco` não são. Para lidar com atributos numéricos no ID3, é preciso dividir os valores em faixas.

Para o atributo `idade`, por exemplo, poderíamos dividir os valores em duas faixas: menor de 40 anos, ou maior de 40 anos. O uso das faixas transforma o atributo em categórico.

O algoritmo de indução de árvores de decisão deve então escolher que testes farão parte da árvore, e em que ordem eles aparecem para cada caminho. Escolher um teste simples é o mesmo que escolher um atributo para ser testado. A questão para construir uma árvore de decisão a partir dos dados é: como escolher os atributos que devem ser testados, e em que ordem?

O algoritmo ID3 usa um critério de *ganho de informação*: em cada etapa na construção da árvore, o próximo teste escolhido deverá ser aquele cuja resposta nos dá a maior quantidade de informação para classificar o item.

A questão agora é saber como medir o ganho de informação de cada teste. Para isso, vamos usar ideias da área conhecida como

Teoria da Informação. Sem entrar nos detalhes da matemática envolvida, vamos usar fórmulas para medir a quantidade de informação que cada teste proporciona, e depois selecionar o teste que fornece o maior ganho.

A quantidade de informação está relacionada à grandeza chamada de *entropia* na Teoria da Informação (não confundir com a entropia da termodinâmica; apesar de haver analogias entre as duas, são duas quantidades diferentes). A entropia para um conjunto de dados é calculada usando a seguinte fórmula:

$$H(S) = -\sum_{x \in X} p_X(x) \log_2 p_X(x)$$

Figura 9.4: Entropia para um conjunto de dados S com classes X

O `S` é o conjunto de dados, `X` é o conjunto de classes, e `p(x)` é a probabilidade de um item ser da classe `x`. No caso do Titanic, o conjunto de classe contém apenas duas: sobrevivência e não-sobrevivência, que chamaremos de `true` e `false`, respectivamente.

Vamos aproximar a probabilidade `p(x)` simplesmente dividindo o número de itens da classe `x` pelo número total de itens. Ou seja, se nosso conjunto de dados de treinamento contém 100 itens (passageiros), sendo 50 de cada classe (50 sobreviventes e 50 não-sobreviventes), a probabilidade *a priori* de um passageiro sobreviver é de 0.5 ou 50%, ou p(`true`) = 0.5.

Essa fórmula nos permite calcular a entropia de um conjunto de dados antes da divisão realizada por um teste. Se aplicarmos um teste a todos os itens do conjunto, isso divide o conjunto de dados original em um número de subconjuntos; esse número é igual ao número de resultados do teste.

Por exemplo, se o teste usa o atributo `embarque`, o conjunto de dados será dividido em três, pois existem três valores possíveis para esse atributo. Medindo a entropia dos três subconjuntos resultantes, o ganho de informação será a diferença entre a entropia do conjunto original e a dos subconjuntos. A entropia dos subconjuntos resultantes do teste por um atributo A é a soma ponderada da entropia de cada subconjunto, e a ponderação utilizada é a proporção de itens no subconjunto em relação ao tamanho do conjunto original.

A equação na figura seguinte resume o cálculo da entropia após a divisão causada por um teste:

$$H_A(S) = \sum_{i=1}^{N} \frac{|S_i|}{|S|} \times H(S_i)$$

Figura 9.5: Entropia para a divisão do conjunto de dados S por um teste do atributo A

Nesta equação, S é o conjunto de dados original, N é o número de valores do atributo A, Si é o subconjunto de dados composto pelos itens que têm o i-ésimo valor para o atributo A, e |S| representa a cardinalidade (número de elementos) do conjunto S.

O ganho de informação obtido em um teste é dado pela diferença da entropia do conjunto original menos a entropia após a divisão. Calculamos o ganho de informação relativo a cada atributo, e selecionamos aquele que fornece o maior ganho.

Repetimos o processo para cada subconjunto resultante do teste selecionado, e continuamos dessa forma, recursivamente, até chegar a um subconjunto em que todos os itens são da mesma classe. Nesse ponto, podemos inserir uma folha na árvore contendo um resultado específico (`true` ou `false`). Este é o algoritmo ID3, e as fórmulas são simples de implementar, como veremos a seguir.

Implementação

As duas principais partes da implementação do algoritmo ID3 são o cálculo do ganho de informação e o algoritmo recursivo de divisão dos dados que gera a árvore. A função `entropia` implementa o cálculo da entropia de um conjunto de dados, de acordo com a equação da figura *Entropia para um conjunto de dados S com classes X*:

```
let entropia s =
  let t, f = contagem_classes s in
  let tf, ff = float t, float f in
  let tfrac = tf /. (tf +. ff) in
  let ffrac = ff /. (tf +. ff) in
  if t = 0 || f = 0 then 0.0
  else -. (tfrac *. log2 tfrac) -. (ffrac *. log2 ffrac)
```

A função `contagem_classes` usada no código apenas conta a ocorrência de sobreviventes e não-sobreviventes no conjunto S:

```
let contagem_classes s =
  let conta (t, f) p =
    if p.sobreviveu then (t+1, f) else (t, f+1)
  in
  List.fold_left conta (0, 0) s
```

A função `entropia` usa o número de sobreviventes (`t`) e o de mortos (`f`) para calcular as frações de itens pertencentes a cada uma das classes, e com isso calcular o valor da entropia. Como não existe logaritmo de zero, assumimos que o valor da entropia é zero caso o conjunto inteiro pertença a uma das classes.

Para calcular a entropia de um conjunto de dados após a partição do conjunto por um teste, usamos a função `entrop_teste` :

```
let entrop_teste s t =
  let part = particao t s in
  let entrop_valor si ac =
    let frac = (float @@ List.length si) /.
               (float @@ List.length s) in
```

```
    ac +. frac *. (entropia si)
  in
  List.fold_right entrop_valor part 0.0
```

A função `entrop_teste` particiona os dados segundo o teste (usando a função `particao`, não mostrada aqui) e calcula a entropia da partição usando a fórmula mostrada na figura *Entropia para a divisão do conjunto de dados S por um teste do atributo A*.

Com a implementação do cálculo dos valores das entropias do conjunto antes e depois da partição causada por um teste, podemos escrever a função que infere uma árvore de decisão a partir de dados. A função `id3` cria uma árvore de decisão para dados de treinamento `d`, usando os testes na lista `lt`; o valor `default` é a classe usada para combinações de valores para as quais o conjunto de treinamento não tem nenhum representante:

```
let id3 d lt default =
  let seleciona_teste d lt =
    let hd = entropia d in
    fst @@ max_f (fun t -> hd -. (entrop_teste d t)) lt
  in
  let rec constroi_arvore lt d =
    match contagem_classes d with
    | (0, 0) -> Result default
    | (n, 0) -> Result true
    | (0, n) -> Result false
    | (t, f) ->
       match lt with
       | [] -> if t > f then Result true else Result false
       | _ ->
          let tix = seleciona_teste d lt in
          let teste = List.nth lt tix in
          let prox_lt = remove_indice tix lt in
          let subs = particao teste d in
          Teste (teste, List.map (constroi_arvore prox_lt) subs)
  in
  constroi_arvore lt d
```

A função interna `seleciona_teste` verifica o ganho de informação de cada teste, e retorna o índice do teste com maior ganho de informação dentro da lista `lt`. A função

`constroi_arvore`, também interna, é a função recursiva que efetivamente constrói a árvore de decisão. Primeiramente, conta-se as classes do conjunto de dados atual; se o conjunto for vazio, usa-se o valor `default`, e se o conjunto de dados for totalmente composto por sobreviventes ou mortos, a função constrói uma folha com o resultado correspondente.

Caso o conjunto atual tenha representantes das duas classes, mas não existam mais testes para usar na árvore, é escolhido como resultado a classe que tiver mais representantes no conjunto de dados atual. Caso ainda existam testes a incluir, o teste com maior ganho de informação é selecionado e removido da
lista de testes, e um nó interno da árvore de decisão é criado. Os filhos desse nó interno serão criados em chamadas recursivas.

Essa função contém algumas ineficiências. Por exemplo, o conjunto atual é particionado duas vezes para o teste selecionado em cada iteração (uma vez para calcular o ganho de informação, outra vez para criar os subconjuntos `subs` para a chamada recursiva). Isso não é difícil de resolver, mas para não complicar o código mostrado aqui, fica como exercício para o leitor.

Com a função `id3`, podemos começar a experimentar com árvores de decisão usando testes de vários atributos. Se incluirmos apenas o teste de gênero, a árvore de decisão construída pelo algoritmo ID3 é idêntica à mostrada na figura *Árvore de decisão que classifica passageiros por gênero*. Para incluir mais atributos, começamos considerando a classe do passageiro, que também indica a classe socioeconômica. O teste para a classe é fácil de escrever:

```
let test_f_classe p =
  match p.classe with
  | Primeira -> 0
  | Segunda -> 1
  | Terceira -> 2
```

```
let teste_classe = {
  nome = "classe";
  f = test_f_classe;
  nvals = 3
}
```

Para facilitar a criação e o uso de árvores para um conjunto de testes e um arquivo de dados de treinamento, escrevemos duas funções utilitárias:

```
let arvore_testes arqtreino testes =
  let d_treino = ler_dados_treino arqtreino in
  id3 d_treino testes false

let classifica_teste_arvore arv arqteste arqsaida =
  let d_teste = ler_dados_teste arqteste in
  let resultado = aplica_arvore_dados arv d_teste in
  escreve_resultado resultado arqsaida
```

Agora podemos criar um programa que lê os dados de treinamento, cria uma árvore de decisão usando os testes para gênero e classe, classifica os dados de teste, e escreve o resultado. Esse programa pode ser visto no arquivo `id3gencls.ml` :

```
let testes = [Titanic.teste_classe; Titanic.teste_genero]

let () =
  Printf.printf
    "Lendo dados de treinamento e construindo arvore\n";
  let arvore = Titanic.arvore_testes "data/train.csv" testes in
  Printf.printf
    "Lendo dados de teste e escrevendo gencls.csv\n";
  Titanic.classifica_teste_arvore arvore
                                  "data/test.csv"
                                  "gencls.csv"
```

A árvore gerada pelo algoritmo ID3 testa primeiro por gênero, depois por classe. Ela prevê que uma mulher deve sobreviver a não ser que pertença à terceira classe, e que homens nunca sobrevivem, independente da classe.

Ao executar o programa e submeter o arquivo de saída gerado

para o Kaggle, obtemos uma pontuação de 0.75598, o que representa uma taxa de acerto de aproximadamente 75.6%. Essa pontuação é mais baixa que o classificador baseado apenas no gênero, o que é interessante: um classificador mais complexo não necessariamente é melhor.

Usando os preços das passagens

Um outro atributo que podemos incluir para aumentar o poder discriminativo do classificador é o preço pago pela passagem, representado pelo atributo `preco`. Esse atributo é numérico e não categórico, possuindo uma grande quantidade de possíveis valores.

Já vimos que o algoritmo ID3 funciona apenas com atributos categóricos, não aceitando atributos numéricos diretamente. Nesse caso, podemos dividir o preço em algumas (poucas) faixas, que se tornam as categorias do atributo. Em vez de considerar diretamente o valor de um preço como 12.0, consideramos que ele pertence à faixa *10 a 20*.

Como a função de teste é arbitrária, podemos incluir essa divisão do preço em faixas diretamente na função de teste, sem precisar fazer outro processamento nos dados de origem. O teste é especificado da seguinte forma:

```
let test_f_preco_4faixas p =
  if p.preco < 10.0 then 0
  else if p.preco < 20.0 then 1
  else if p.preco < 30.0 then 2
  else 3

let teste_preco_4faixas = {
  nome = "preco em 4 faixas";
  f = test_f_preco_4faixas;
  nvals = 4
}
```

Com esse teste, escrevemos um novo programa que treina uma

árvore de decisão usando gênero, classe e preço, e depois aplica essa árvore nos dados de teste. Esse programa é idêntico ao que classifica por gênero e classe, apenas com a adição do novo teste. O código-fonte do programa está no arquivo `id3prgencls.ml` :

```
let testes = [
  Titanic.teste_classe;
  Titanic.teste_genero;
  Titanic.teste_preco_4faixas
]

let () =
  Printf.printf
    "Lendo dados de treinamento e construindo arvore\n";
  let arvore = Titanic.arvore_testes "data/train.csv" testes in
  Printf.printf
    "Lendo dados de teste e escrevendo prgencls.csv\n";
  Titanic.classifica_teste_arvore arvore
                                  "data/test.csv"
                                  "prgencls.csv"
```

Executando esse programa e submetendo o arquivo gerado no Kaggle, obtemos uma pontuação de 0.77990, ou seja, aproximadamente 78% de acerto. Essa pontuação é idêntica a um classificador mostrado no tutorial do Kaggle construído para os mesmos três atributos. A diferença é que no tutorial isso foi feito manualmente, analisando os dados para cada combinação dos três atributos e escolhendo uma classe para cada combinação baseado nas taxas de sobrevivência observadas.

Aqui, criamos um classificador de mesmo desempenho apenas especificando os testes e usando o algoritmo ID3 para gerar uma árvore de decisão automaticamente. Isso torna muito mais fácil a experimentação com mais atributos ou mesmo com formas diferentes de dividir um atributo numérico como o preço.

As pontuações das três estratégias mostradas neste capítulo podem ser vistas na figura:

Submission	Files	Public Score	Selected?
Sat, 11 Oct 2014 22:23:37 Árvore de decisão criada com ID3, com testes p ara gênero, classe e 4 faixas de preço. Edit description	prgencls.csv	0.77990	☐
Sat, 11 Oct 2014 02:08:21 Classificação usando gênero e classe, com árvo re aprendida via ID3 Edit description	gencls.csv	0.75598	☐
Tue, 23 Sep 2014 02:15:29 Modelo de genero, em OCaml Edit description	genero.csv	0.76555	☐

Figura 9.6: Pontuações dos três classificadores desenvolvidos neste capítulo

A partir dessa base já pronta para o problema do Titanic, muitas melhorias podem ser feitas. Pode-se experimentar com o uso de outros atributos, ou dividir o preço da tarifa em mais faixas, por exemplo. Podemos cogitar a possibilidade de outros atributos que influenciam a sobrevivência; por exemplo, talvez a presença de filhos no navio aumente as chances de sobrevivência do passageiro.

Para experimentar com isso, é só incluir um teste para o atributo `paisfilhos` . Como esse atributo é numérico, é preciso fazer um teste que divide os valores em faixas. Esse teste pode ser criado rapidamente e aproveita toda a infraestrutura já mostrada neste capítulo.

Uma possibilidade mais avançada seria melhorar a representação das árvores de decisão, como já discutido. O uso de metaprogramação pode facilitar a criação e uso de árvores, e permitir uma representação que evita estados ilegais.

Outra direção mais trabalhosa seria mudar a estratégia de classificação de dados. Usar um algoritmo mais avançado de criação de árvores, como C4.5, pode resultar em classificadores melhores.

Indo mais longe, pode-se testar outras técnicas de classificação, como árvores aleatórias, redes neurais, SVMs (*Support Vector Machines*), e classificadores de *ensemble* que combinam vários outros classificadores. Muitas outras técnicas estão disponíveis na literatura de aprendizado de máquina e análise de dados.

Capítulo 10

PARÂMETROS ROTULADOS

A aplicação de funções é uma operação básica na programação funcional, e funções são aplicadas a parâmetros. Quando a complexidade das funções e do código que as chama aumenta, é interessante ter possibilidades que tornem mais prática a passagem de parâmetros.

Características como parâmetros nomeados e opcionais estão presentes em muitas outras linguagens de programação. Neste capítulo, veremos como a linguagem OCaml provê essas possibilidades através do uso de *parâmetros rotulados*.

10.1 RÓTULOS PARA NOMEAR PARÂMETROS

Funções que possuem muitos parâmetros, ou para as quais não é clara a ordem deles, podem ser mais práticas de usar se eles forem nomeados. Um exemplo são os *folds*: sabemos que são três parâmetros, sendo uma função de redução, uma lista de elementos, e um valor inicial para a redução. Entretanto, a ordem entre eles não é clara, e varia de acordo com a direção do *fold*. Essa ordem pode ser facilmente verificada usando o REPL:

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Ambas recebem a função como primeiro parâmetro, mas `fold_left` recebe o valor inicial como segundo, e a lista como último, enquanto em `fold_right` essa ordem dos dois últimos é invertida.

Podemos criar funções de redução usando parâmetros rotulados colocando um caractere `~` antes do nome de cada um:

```
# let foldl ~f ~ini ~lista =
    List.fold_left f ini lista;;
val foldl : f:('a -> 'b -> 'a) -> ini:'a ->
            lista:'b list -> 'a = <fun>
```

Observando o tipo inferido pelo REPL, vemos que os rótulos dos parâmetros aparecem nesse tipo. Isso também ajuda a documentar a função, já que o tipo das funções é mostrado em vários lugares. Para chamar uma função com argumentos rotulados, usamos o rótulo do parâmetro, seguido de um caractere `:` e o valor:

```
# foldl ~f:(+) ~ini:0 ~lista:[1; 2; 3; 4; 5];;
- : int = 15
```

Com os argumentos rotulados, a ordem de passagem dos parâmetros não importa mais:

```
# foldl ~ini:0 ~f:(+) ~lista:[1; 2; 3; 4; 5];;
- : int = 15
# foldl ~lista:[1; 2; 3; 4; 5] ~f:(+) ~ini:0;;
- : int = 15
```

Outra vantagem do uso de rótulos é a possibilidade de aplicar parcialmente qualquer parâmetro nomeado, independente da ordem. Por exemplo, `List.map` recebe como parâmetros uma função de mapeamento e uma lista, nesta ordem. Podemos aplicar `List.map` parcialmente, passando apenas a função de mapeamento, e usar esse mapeador de função fixa:

```
# let mapquad = List.map (fun x -> x * x);;
val mapquad : int list -> int list = <fun>
```

```
# mapquad [14; 42];;
- : int list = [196; 1764]
# mapquad [7; 3; 5];;
- : int list = [49; 9; 25]
```

Porém, não podemos aplicar `List.map` parcialmente a uma lista, e depois mapeá-la com várias funções:

```
# let map123 = List.map [1; 2; 3];;
Error: This expression has type 'a list
       but an expression was expected of type 'b -> 'c
```

Se criarmos um `map` com argumentos rotulados, ambas as formas de aplicação parcial são possíveis:

```
# let map ~f ~lista =
    List.map f lista;;
val map : f:('a -> 'b) -> lista:'a list -> 'b list = <fun>

# let mapquad = map ~f:(fun x -> x * x);;
val mapquad : lista:int list -> int list = <fun>
# mapquad [14; 42];;
- : int list = [196; 1764]
# mapquad [7; 3; 5];;
- : int list = [49; 9; 25]

# let map123 = map ~lista:[1; 2; 3];;
val map123 : f:(int -> 'a) -> 'a list = <fun>
# map123 (fun x -> x + 2);;
- : int list = [3; 4; 5]
# map123 (fun x -> x * 2);;
- : int list = [2; 4; 6]
```

Na maioria dos casos, é possível chamar uma função com parâmetros rotulados sem os rótulos se todos os parâmetros forem passados na chamada, e estiverem na ordem de declaração (a ordem mostrada no tipo da função):

```
# map (fun x -> x * x) [1; 2; 3; 4];;
- : int list = [1; 4; 9; 16]
```

Quando o tipo de retorno de uma função é uma variável de tipo (ou seja, um tipo completamente desconhecido), não é possível aplicar a função sem os rótulos. Em alguns casos, a tentativa de

chamá-la sem os rótulos resulta em erros de tipo, em outros pode ocorrer um resultado inesperado. Isso acontece, por exemplo, com a função `foldl` vista antes:

```
# foldl;;
- : f:('a -> 'b -> 'a) -> ini:'a -> lista:'b list -> 'a = <fun>
# foldl ~f:(+) ~ini:0 ~lista:[1; 2; 3; 4; 5];;
- : int = 15
# foldl (+) 0 [1; 2; 3; 4; 5];;
- : f:(((int -> int -> int) -> int -> int list -> 'a) ->
       'b -> (int -> int -> int) -> int -> int list -> 'a) ->
    ini:((int -> int -> int) -> int -> int list -> 'a) ->
    lista:'b list -> 'a
= <fun>
```

A tentativa de chamar `foldl` sem os rótulos resulta em uma função com tipo bastante complicado. Isso ocorre por causa da interação entre os rótulos (que fazem parte do tipo da função), o resto do sistema de tipos, e a inferência de tipos da linguagem.

Nesse caso, `foldl` foi considerada como aplicada parcialmente, e por isso ela retorna uma outra função. Esse problema só acontece quando o tipo de retorno é *apenas* uma variável de tipo como `'a`, não apenas quando existe uma variável de tipo no retorno. Por exemplo, a função `map` que criamos antes tem tipo de retorno `'b list`:

```
# map;;
- : f:('a -> 'b) -> lista:'a list -> 'b list = <fun>
```

Mas ela pode ser aplicada sem os rótulos, como já vimos. A propósito, as duas funções que criamos já existem na biblioteca padrão: o módulo `ListLabels` tem as mesmas funções que o módulo `List`, mas usando parâmetros rotulados:

```
# ListLabels.fold_left;;
- : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a = <fun>
# ListLabels.map;;
- : f:('a -> 'b) -> 'a list -> 'b list = <fun>
```

Em ambas as funções existem parâmetros rotulados e não

rotulados. Usar rótulos apenas para alguns parâmetros é uma prática comum em OCaml, e para entender como funciona a mistura de parâmetros rotulados e não rotulados, precisamos entender melhor como funcionam os rótulos.

Rótulos e nomes de parâmetros

Apesar do que pode parecer até agora, os rótulos e nomes de parâmetros não são a mesma coisa. A forma completa de declarar um parâmetro rotulado é `~rot:par`, em que `rot` é o rótulo e `par` é o nome do parâmetro, o nome de variável que pode ser usado na função. Por exemplo:

```
# let f ~x:x1 ~y:y1 = x1 * y1;;
val f : x:int -> y:int -> int = <fun>

# f ~x:3 ~y:9;;
- : int = 27
```

Nesse exemplo, os rótulos são `x` e `y`, como dá para ver na chamada à função `f`, mas os nomes deles são `x1` e `y1`, como visto na definição da função `f`. O rótulo `x` é usado para designar o valor do parâmetro `x1`, e o rótulo `y` é usado para designar o valor do parâmetro `y1`.

Um parâmetro especificado apenas com um rótulo `~r` é um atalho para `~r:r`, que usa o mesmo nome para o rótulo e o parâmetro. Essa diferença entre rótulos e parâmetros é importante para entender o mecanismo em sua generalidade.

Vários parâmetros podem ser declarados com o mesmo rótulo. Neste caso, os parâmetros são diferenciados pela ordem:

```
# let g ~x:x1 ~x:x2 ~y:y1 = x1 * y1 + x2;;
val g : x:int -> x:int -> y:int -> int = <fun>
# g ~x:3 ~x:5 ~y:2;;
- : int = 11
```

Em chamadas para a função `g`, o primeiro parâmetro rotulado

com `~x` será o valor de `x1`, e o segundo será o valor de `x2`. Mas a ordem entre os parâmetros com rótulos diferentes não importa:

```
# g ~x:3 ~y:2 ~x:5;;
- : int = 11
```

Isso é importante porque parâmetros que não são declarados com rótulos são considerados associados a um rótulo vazio. Portanto, todos os que são declarados sem rótulo são associados a um mesmo rótulo; a única forma de diferenciar entre eles é pela ordem.

É pouco comum usar vários com o mesmo rótulo, mas a maioria das funções com parâmetros rotulados mistura parâmetros com e sem rótulo. Os rotulados podem ser especificados em qualquer ordem, mas os não rotulados devem obedecer à ordem de declaração entre eles. As três chamadas para a função `h` são equivalentes:

```
# let h ~x y z = x * y + z;;
val h : x:int -> int -> int -> int = <fun>

# h ~x:3 2 5;;
- : int = 11
# h 2 ~x:3 5;;
- : int = 11
# h 2 5 ~x:3;;
- : int = 11
```

Os casos mais comuns vão usar um ou dois parâmetros rotulados, e o resto serão parâmetros posicionais (sem rótulo). Ter um argumento rotulado aumenta a flexibilidade para as chamadas de função e também para a aplicação parcial de parâmetros, como já vimos. Em uma função que mistura rotulados e não rotulados, parâmetros passados em uma aplicação só são associados a parâmetros rotulados se o rótulo for especificado explicitamente, ou se todos forem especificados. Por exemplo, para a função `h` criada antes:

```
# let h2 = h 2;;
val h2 : x:int -> int -> int = <fun>
# let h25 = h 2 5;;
val h25 : x:int -> int = <fun>
```

No primeiro caso, o número `2` é associado ao parâmetro `y` (o primeiro sem rótulo). No segundo, `2` é o valor de `y` e `5` é o valor do parâmetro `z`. Nos dois casos, a função resultante da aplicação parcial espera um parâmetro rotulado com rótulo `x`. No caso da função `h25`, ele pode ser passado com ou sem rótulo, porque só há um parâmetro:

```
# h25 ~x:3;;
- : int = 11
# h25 3;;
- : int = 11
```

Para aplicar parcialmente a função `h` ao parâmetro `x`, é preciso usar o rótulo:

```
# let h3 = h ~x:3;;
val h3 : int -> int -> int = <fun>
# h3 2 5;;
- : int = 11
```

Outra característica desejável para a passagem de parâmetros é ter parâmetros *opcionais*, que também são suportados em OCaml.

10.2 PARÂMETROS OPCIONAIS

O mecanismo de rótulos também possibilita o uso de parâmetros opcionais, que podem ou não ser passados em cada chamada. Um parâmetro opcional é indicado com um rótulo começando com uma interrogação em vez de um `~`.

Um valor *default* para o parâmetro opcional pode ser especificado. Para exemplificar, vamos escrever uma função que calcula a raiz enésima de um número. Faremos o índice da raiz opcional, com valor *default* igual a dois, já que a raiz quadrada é o

caso mais comum:

```
# let raiz ?(n=2) x =
    x ** (1.0 /. (float n));;
val raiz : ?n:int -> float -> float = <fun>
```

O rótulo `n` e o fato de ele ser opcional aparecem no próprio tipo da função. O caso comum pode ser usado sem especificar o índice, e outros índices podem ser usados se o parâmetro opcional for passado. Na chamada da função, a passagem de um parâmetro opcional ocorre da mesma forma que um rotulado não opcional:

```
# raiz 25.0;;
- : float = 5.

# raiz 27.0;;
- : float = 5.19615242270663202

# raiz ~n:3 27.0;;
- : float = 3.
```

Uma função com parâmetros opcionais deve ter pelo menos um parâmetro não opcional para que possa ser omitido; o não opcional deve vir depois do opcional na ordem, pois é a presença de um valor para o parâmetro não opcional na chamada da função que indica que o opcional foi omitido. Podemos criar uma função apenas com um parâmetro opcional, mas ele nunca pode ser omitido, e sempre deve ser passado com rótulo:

```
# let f ?(x=1.0) = x *. x;;
val f : ?x:float -> float = <fun>
Characters 6-23:
Warning 16: this optional argument cannot be erased.

# f;;
- : ?x:float -> float = <fun>

# f 2.0;;
Error: The function applied to this argument has type
       ?x:float -> float
This argument cannot be applied without label

# f ~x:2.0;;
```

```
- : float = 4.
```

Se só houver um parâmetro não opcional e ele for rotulado como um parâmetro rotulado, na chamada de função é independente de ordem. A omissão do parâmetro opcional só será detectada se a chamada não usar um rótulo:

```
# let opcional_rot ?(x=1.0) ~y =
    x *. y;;
val opcional_rot : ?x:float -> y:float -> float = <fun>

# opcional_rot 3.4;;
- : float = 3.4

# opcional_rot ~x:2.0 ~y:2.3;;
- : float = 4.6

# opcional_rot ~y:2.3;;
- : ?x:float -> float = <fun>
```

Chamadas à função `opcional_rot` que omitem um valor para `x` não podem usar o rótulo `y`, como visto no último exemplo.

Valores default

Um parâmetro opcional sem valor *default* recebe um tipo opcional na inferência. Se a chamada à função não passa um valor para o parâmetro opcional, este terá valor `None` dentro da função; se passar um valor `v`, o seu valor dentro da função será `Some v`.

A função a seguir calcula o recíproco de um número `den`, calculando a divisão de `num` por `den`. Se o parâmetro `num` não for passado, usa-se o numerador igual a 1:

```
let recip ?num den =
  match num with
  | Some n -> n /. den
  | None -> 1.0 /. den
```

Esse código é muito mais simples usando um valor *default*, que é equivalente:

```
let recip ?(num=1.0) den =
  num /. den
```

Mas em alguns casos, não existe um valor *default* adequado, e o uso de um tipo opcional é necessário.

Uma aplicação de função que omite parâmetros opcionais não é uma aplicação parcial e não pode ser aplicada aos parâmetros opcionais posteriormente. Por exemplo, para a função `recip`:

```
# recip 25.0;;
- : float = 0.04

# recip ~num:2.0 25.0;;
- : float = 0.08

# (recip 25.0) ~num:2.0;;
Error: This expression has type float
       This is not a function; it cannot be applied.
```

No último caso, ao omitir o parâmetro `num`, a aplicação ficou completa usando o valor *default*. Portanto, `recip 25.0` não é uma aplicação parcial e não pode ser aplicada a mais um parâmetro.

O uso de parâmetros rotulados também pode ter um impacto negativo em combinação com duas características importantes da linguagem: inferência de tipos e funções de alta ordem.

10.3 INFERÊNCIA DE TIPOS E FUNÇÕES DE ALTA ORDEM

Os parâmetros rotulados tornam mais conveniente a passagem de parâmetros para funções, mas isso vem ao custo de maior complexidade em dois pontos importantes da linguagem: inferência de tipos e funções de alta ordem.

Como os rótulos associados aos parâmetros fazem parte do tipo das funções, a inferência de tipos para código usando rótulos nem sempre funciona de maneira tão prática e correta quanto acontece

com código sem rótulos. A ordem dos rótulos é um dos possíveis problemas. Vejamos as duas funções a seguir:

```
# let f ~x ~y = x * y;;
val f : x:int -> y:int -> int = <fun>
# let h g = g ~y:2 ~x:3;;
val h : (y:int -> x:int -> 'a) -> 'a = <fun>
```

A função `f` recebe dois parâmetros rotulados, com rótulos `x` e `y`. A função `h` recebe como parâmetro uma função `g` que também tem dois parâmetros rotulados, com rótulos de mesmo nome que os de `f`. Como os parâmetros rotulados podem ser passados em qualquer ordem, poderíamos esperar que seria possível passar a função `f` como parâmetro para `h`, mas isso não acontece:

```
# h f;;
Error: This expression has type x:int -> y:int -> int
       but an expression was expected of type
       y:int -> x:int -> 'a
```

Os tipos de `f` e do parâmetro `g` de `h` não são compatíveis por causa da ordem dos rótulos. A ordem dos parâmetros rotulados na chamada de função não é importante, mas a ordem nos tipos sim. Apesar de o parâmetro `g` de `h` poder ter tipo igual a `f`, pela forma como a função `h` foi escrita (passando `y` e depois `x`), o tipo inferido tem uma ordem diferente.

Uma maneira de resolver isso seria declarando explicitamente o tipo de `g`:

```
# let h (g: x:int -> y:int -> int) = g ~y:2 ~x:3;;
val h : (x:int -> y:int -> int) -> int = <fun>
# h f;;
- : int = 6
```

Porém, ter de declarar o tipo do parâmetro é uma indicação de que o sistema de inferência de tipos não funcionou nessa situação. Outro problema que pode acontecer com a inferência de tipos é a diferença entre um parâmetro opcional e um rotulado não opcional. Se um parâmetro for passado com rótulo para uma função, a

inferência de tipos não vai considerá-lo opcional, e os dois tipos de parâmetros são diferentes.

A função `intervalo` constrói uma lista com todos os números dentro de um intervalo; o incremento é um parâmetro opcional:

```
# let intervalo ?(inc = 1) min max =
    let rec loop i =
      if i > max then [] else i :: loop (i + inc)
    in
    loop min;;
val intervalo : ?inc:int -> int -> int -> int list = <fun>

# intervalo 1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

# intervalo ~inc:2 2 16;;
- : int list = [2; 4; 6; 8; 10; 12; 14; 16]
```

Se tentarmos agora escrever uma função que testa outras funções que possuem um parâmetro rotulado chamado `inc`, ela não vai funcionar com `intervalo`:

```
# let teste_inc r = r ~inc:2 2 20;;
val teste_inc : (inc:int -> int -> int -> 'a) -> 'a = <fun>

# test_inc intervalo;;
Error: This expression has type
         ?inc:int -> int -> int -> int list
       but an expression was expected of type
         inc:int -> int -> int -> 'a
```

Como podemos ver no erro mostrado, a fonte de incompatibilidade entre os tipos do parâmetro de `teste_inc` e `intervalo` é o primeiro argumento ser opcional, ou rotulado mas não opcional. A inferência não considerou que o parâmetro `inc` de `r` é opcional. Mais uma vez, isso pode ser consertado declarando explicitamente um tipo para o parâmetro `r` da função `teste_inc`.

Os problemas exemplificados até agora acontecem principalmente quando se usa funções com parâmetros rotulados

como parâmetros para funções de alta ordem. O uso de rótulos em funções passadas como parâmetros pode causar incompatibilidades inesperadas. Isso é fácil de ver:

```
# let h g = g ~x:3 ~y:2;;
val h : (x:int -> y:int -> 'a) -> 'a = <fun>
# let f x y = x * y;;
val f : int -> int -> int = <fun>

# h f;;
Error: This expression has type int -> int -> int
  but an expression was expected of type x:int -> y:int -> 'a
```

A função `f` não pode ser passada como parâmetro de `h` por causa dos rótulos, apesar de os tipos dos parâmetros parecerem os mesmos. Na verdade, os tipos não são os mesmos, pois os rótulos fazem parte do tipo, e são comparados como parte dele: o tipo `x:int` é `int` com rótulo `x`, e o tipo `int` (como parâmetro de uma função) é `int` com rótulo vazio. Como o rótulo vazio não é igual ao `x`, os dois tipos são diferentes.

Existe uma grande quantidade de código na linguagem OCaml que usa parâmetros rotulados, opcionais ou não, e os erros mostrados nessa seção só acontecem raramente. Entretanto, é importante ter conhecimento dos problemas que podem acontecer.

Essas armadilhas podem causar dúvidas na hora de escolher usar parâmetros rotulados. Para reduzir as dúvidas, vamos discutir algumas indicações de quando e como usar os parâmetros rotulados.

Um equilíbrio delicado

O sistema de tipos usado como base da linguagem OCaml (e outras linguagens da família ML, além da linguagem Haskell) é conhecido como *sistema Hindley-Milner*. Ele é considerado como tendo um ótimo equilíbrio entre expressividade e conveniência: os tipos no sistema H-M são bastante expressivos (mais expressivos que os de muitas linguagens famosas como Java) e ao mesmo tempo a inferência de tipos funciona extremamente bem para eles.

Algumas décadas de pesquisa (as bases do sistema H-M foram estabelecidas na década de 1970) mostraram que praticamente qualquer tentativa de estender a expressividade do sistema de tipos H-M resulta em menor conveniência, pois tipos mais expressivos são mais difíceis de inferir, e a inferência não funciona tão bem na presença desses tipos. Os rótulos vistos neste capítulo são um exemplo.

A linguagem Haskell implementada pelo compilador GHC possui um grande número de extensões que tornam o sistema de tipos da linguagem muito mais expressivo, mas que raramente podem ser inferidos automaticamente. Sair de perto do ponto ótimo que é o sistema Hindley-Milner sempre tem um custo.

10.4 SUGESTÕES PARA O BOM USO DE RÓTULOS

Rótulos podem ser extremamente úteis e convenientes quando bem usados, porém, como vimos, também possuem algumas

armadilhas. Para evitá-las, vale a pena seguir algumas recomendações de uso.

O próprio manual da linguagem indica que usar rótulos é positivo quando torna o programa mais legível, é fácil de lembrar, e permite aplicações parciais interessantes sempre que possível. Essas ideias são vagas, mas dão uma ideia do que buscar.

Em geral, não se usa rótulos em todos os parâmetros de uma função, e em muitos casos usa-se apenas para parâmetros opcionais. Os rótulos funcionam melhor quando utilizados com parcimônia, e essas sugestões do manual servem como um guia para ajudar a decidir quando usar.

A biblioteca padrão OCaml tem alguns módulos como `List` e `ListLabels` que possuem as mesmas funções, em versões com e sem rótulos. Isso acontece porque os rótulos foram adicionados à linguagem na versão 3 e, para manter a compatibilidade, os rótulos não foram adicionados às funções já existentes. Bibliotecas criadas mais recentemente, como a Core da empresa Jane Street Capital (que pretende substituir a biblioteca padrão), usam rótulos com frequência.

Um dos critérios usados na biblioteca padrão OCaml é não utilizar rótulo para o parâmetro principal de uma função. Para muitas funções, é simples saber qual o parâmetro principal. Vejamos o exemplo da função `fold_right` do módulo `ListLabels`:

```
# List.fold_right;;
- : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b = <fun>
```

Para as funções do módulo `ListLabels`, o parâmetro principal é uma lista. Pelo tipo da função `fold_right`, vemos que a lista passada como parâmetro para `ListLabels` não é rotulada:

```
# module List = ListLabels;;
```

```
module List = ListLabels

# List.fold_right ~f:(+) ~init:0 [1; 2; 3; 4; 5];;
- : int = 15
```

Nesse exemplo, criamos um módulo `List` para ser um sinônimo do módulo `ListLabels`. Isso esconde o módulo `List` original (sem rótulos) e nos permite chamar as funções rotuladas de maneira mais compacta. O fato de a lista passada como parâmetro não ser rotulada ajuda na composição de funções.

Se o parâmetro de uma função é outra função, ele pode ser rotulado (como acontece em `fold_right`), mas os parâmetros da função passada como parâmetro não devem possuir rótulos. A presença de rótulos no tipo de um parâmetro função impede o uso de funções cujos parâmetros não têm rótulos, como vimos antes:

```
# let h1 g = g ~x:3 ~y:2;;
val h1 : (x:int -> y:int -> 'a) -> 'a = <fun>

# let f1 x y = x * y;;
val f1 : int -> int -> int = <fun>

# h1 f1;;
Characters 3-5:
  h1 f1;;
     ^^
Error: This expression has type int -> int -> int
       but an expression was expected of type
       x:int -> y:int -> 'a
```

O parâmetro `g` da função `h1` é uma função que tem parâmetros rotulados. Logo, apenas funções com exatamente os mesmos rótulos pode ser passada para `h1`. Isso é uma limitação que reduz as possibilidades de composição para a função `h1`. É possível criar uma função para compatibilizar os tipos com e sem rótulo, por exemplo:

```
# let f1_com ~x ~y = f x y;;
val f1_com : x:int -> y:int -> int = <fun>

# h1 f1_com;;
```

```
- : int = 6
```

Mas o ideal é evitar a necessidade de criar esse tipo de função *wrapper*.

Como mencionado nas dicas do manual, ter rótulos fáceis de lembrar é importante. Uma das motivações para usar parâmetros rotulados é não precisar lembrar com toda precisão a ordem dos parâmetros para uma função que recebe vários, como `fold_right`. Rotular os parâmetros pode nos permitir chamar a função sem precisar consultar a documentação, ou usar o REPL para mostrar o tipo da função.

Mas se não lembramos dos nomes dos rótulos dos parâmetros, teremos de consultar a documentação ou o REPL da mesma forma. A sugestão para evitar esse problema é usar nomes padronizados para os rótulos de parâmetros não opcionais. As sugestões de nomes mostradas no manual da linguagem são:

- `f` para funções que serão aplicadas;
- `pos` para posições em *arrays* ou outros contêineres;
- `len` para tamanhos (do inglês `length`);
- `buf` para *buffers* (geralmente do tipo `bytes`);
- `src` para o ponto inicial (*source*) de uma operação (por exemplo, cópia);
- `dst` para o ponto de destino de uma operação (por exemplo, cópia);
- `init` para o valor inicial de um iterador;
- `cmp` para uma função de comparação.

Como é normal no aprendizado de linguagens de programação, é a experiência com a linguagem OCaml e a leitura de código escrito por outros que mais ajuda a entender as nuances de quando usar e quando não usar rótulos. Também vale a pena consultar o tutorial do manual oficial sobre rótulos, que pode ser encontrado no

endereço: http://caml.inria.fr/pub/docs/manual-ocaml/lablexamples.html.

Capítulo 11

VARIANTES POLIMÓRFICAS E EXTENSÍVEIS

Os tipos variantes ou tipos de dados algébricos (ADTs) da linguagem OCaml foram introduzidos no capítulo *Registros e variantes*, e usados extensivamente no resto do livro. Os ADTs são uma característica central nas linguagens funcionais com tipagem estática e, juntamente com a capacidade de *pattern matching*, formam uma combinação extremamente produtiva.

Mesmo assim, existem alguns fatores dos ADTs tradicionais que limitam sua utilidade. As variantes polimórficas são uma versão estendida dos tipos variantes, com novas capacidades que os tornam mais práticos de usar em algumas situações. Assim como acontece com os rótulos, esse poder adicional das variantes polimórficas tem um custo, mas é um que vale à pena pagar em vários casos.

Outra adição recente dos ADTs em OCaml são os tipos variantes extensíveis, que admitem a criação de tipos variantes *abertos*. Novas variantes podem ser adicionadas a um tipo extensível a qualquer momento (de forma similar à declaração de novas exceções).

Neste capítulo, vamos examinar as variantes polimórficas e os tipos variantes extensíveis na linguagem OCaml, analisando

também as limitações dessas duas características.

11.1 LIMITAÇÕES DOS TIPOS VARIANTES

Um problema com os ADTs tradicionais é que cada construtor só pode ser usado em um tipo, e esse tipo não pode ser estendido ou restrito (ou seja, construtores não podem ser retirados). Mas existem casos em que um dado precisa convertido de uma representação para outra, e a representação de destino é uma versão estendida (ou restrita) da representação de origem. Com ADTs tradicionais, todas as possibilidades de fazer isso possuem características insatisfatórias.

Vamos tornar a discussão mais concreta com um exemplo. Digamos que nossa aplicação deve ler ou receber dados no formato JSON (http://json.org/), processá-los, e escrever ou enviar os resultados no formato BSON (http://bsonspec.org/). JSON é um formato de representação de dados em texto, enquanto BSON é um formato binário para serialização de dados. O formato BSON foi criado como uma extensão do JSON, com algumas possibilidades adicionais.

O formato JSON pode representar dois tipos de estruturas: objetos e listas (que também podem ser considerados vetores ou *arrays*). Um objeto é uma sequência de pares `chave : valor`, em que a chave é uma *string* e o valor é qualquer valor JSON. Uma lista é uma sequência de valores JSON.

Um valor JSON pode ser um número, uma *string*, um objeto, uma lista, ou as constantes `true`, `false` e `null`. O exemplo a seguir mostra um objeto no formato JSON, representando uma entrada no histórico de navegação web de um usuário:

```
{
    "url": "http://xkcd.com",
```

```
    "descricao": "Quadrinho XKCD",
    "acessos": 22,
    "ultimo": "2014-10-15 14:17:22"
}
```

A entrada inclui a URL acessada, uma descrição do site, o número de acessos no último mês, e a data do último acesso. A data do último acesso é representada como uma *string*.

Um ADT para representar um valor JSON poderia ser escrito da seguinte forma:

```
type json =
| String of string
| Num of float
| Null
| Bool of bool
| Lista of json list
| Obj of (string * json) list
```

No tipo `json`, as constantes `true` e `false` são representadas com o construtor `Bool`. O construtor `Lista` representa uma lista de valores JSON, e o construtor `Obj` representa um objeto como uma lista de pares de `string` e valor `json`. A entrada no histórico de navegação mostrada no formato JSON seria representado pelo tipo `json` da seguinte forma:

```
let hist1 =
  Obj [("url", String "http://xkcd.com");
       ("descricao", String "Quadrinho XKCD");
       ("acessos", Num 22.0);
       ("ultimo", String "2014-10-15 14:17:22")]
```

O formato BSON é binário, mas foi criado baseado no JSON. A maior parte das possibilidades para um valor BSON é idêntica aos valores JSON, mas existe suporte para novos tipos de valores, como data e *timestamp*. Por outro lado, o formato BSON não suporta valores do tipo número.

Para representar valores JSON e BSON na mesma aplicação, usando ADTs tradicionais, temos algumas opções. Para evitar

repetição e ter de copiar valores idênticos, poderíamos pensar em definir um tipo só para os dois formatos, contendo construtores para todos os tipos de valores possíveis nos dois.

Essa ideia fere um princípio que já discutimos antes: esse tipo incluindo todos os valores permite a representação de estados ilegais. Uma função que processa um valor JSON deveria recusar um valor do tipo data, por exemplo, gerando uma exceção ou outro tipo de erro. As funções que processam apenas valores BSON também teriam de recusar valores de tipo número. Isso seria verificado dinamicamente, em tempo de execução. O resultado final é uma aplicação que pode ter erros similares a erros de tipo, e perde a oportunidade de verificar a adequação dos valores em tempo de compilação (usando tipos).

Uma opção mais correta seria definir dois tipos: um para valores JSON (como o tipo `json` já mostrado) e um para valores BSON. Isso é melhor do que permitir a representação de valores ilegais, mas também tem algumas desvantagens.

Uma delas é que os dois tipos não podem usar os mesmos construtores, ou seja, um construtor como `String` não pode fazer parte de dois tipos. Uma solução seria usar um prefixo diferente para cada tipo, por exemplo: construtores `JString`, `JLista` etc., para JSON; e `BString`, `BLista` etc., para BSON. Melhor ainda, podemos usar módulos diferentes para os dois tipos (usando a convenção de chamar de `t` o tipo principal de um módulo):

```
module JSON = struct
  type t =
  | String of string
  | Num of float
  | Null
  | Bool of bool
  | Lista of t list
  | Obj of (string * t) list
end
```

```
module BSON = struct
  type t =
  | String of string
  | Null
  | Bool of bool
  | Lista of t list
  | Obj of (string * t) list
  | Data of int
end
```

Agora os construtores podem ser acessados com o prefixo do nome do módulo: `JSON.Lista` e `BSON.Lista`. O tipo para um valor BSON é bastante simplificado com relação à especificação oficial, mas demonstra a parte importante do nosso problema aqui: ele contém o construtor `Data` que o tipo `JSON.t` não tem, e não contém o construtor `Num`. A data é representada como um número inteiro que codifica a data e a hora contando o número de milissegundos desde a data inicial do Unix (01/01/1970).

Um problema com esses tipos aparece quando é necessário transformar um valor JSON em BSON (ou vice-versa). A maioria dos construtores nos dois tipos é idêntica, mas os tipos são diferentes. É preciso criar uma função que, para a maioria dos casos, simplesmente copia um construtor para outro idêntico no outro tipo, e faz uma transformação interessante apenas nos casos diferentes.

Para os tipos criados anteriormente para valores JSON e BSON, isso não chega a ser um problema tão grande. Mmas se os tipos tivessem mais construtores, ou fosse preciso transformar um valor de um tipo para vários tipos parecidos (mas não idênticos), essa repetição toda se tornaria, no mínimo, tediosa de escrever.

O ideal seria poder compartilhar variantes entre os dois tipos, porém mantendo dois tipos diferentes que possuem algumas das mesmas variantes. Isso é possível de fazer com variantes polimórficas, como veremos a seguir.

11.2 VARIANTES POLIMÓRFICAS

Uma variante polimórfica é similar a um construtor ou variante de um ADT tradicional, mas o nome da variante polimórfica deve começar com o caractere *backtick* (`), normalmente seguido por uma letra maiúscula. Não é preciso declarar uma variante polimórfica antes de usá-la:

```
# `Null;;
- : [> `Null ] = `Null
```

O tipo mostrado para a variante polimórfica `` `Null `` é interessante: *[> `Null]*. Isso significa que a variante `` `Null `` pertence a um tipo que inclui uma variante `` `Null ``, mas talvez outras possibilidades (isso é representado pelo caractere `>`).

Um tipo de variantes polimórficas é sempre uma lista de variantes suportadas, opcionalmente começando com um caractere `>` quando o tipo assume que outras variantes são possíveis, ou com um caractere `<` quando no máximo essas variantes são parte do tipo. Um exemplo é se criamos uma função que faz um `match` com variantes polimórficas. A função a seguir transforma alguns valores JSON em *strings*:

```
# let valor_string v =
    match v with
    | `Null -> "(null)"
    | `Num n -> string_of_float n
    | `String s -> s;;
val valor_string :
  [< `Null | `Num of float | `String of bytes ] -> bytes = <fun>
```

Antes de analisar a função, vale a pena observar que variantes polimórficas podem guardar valores, assim como as tradicionais:

```
# `Num 33.2;;
- : [> `Num of float ] = `Num 33.2
```

Voltando para a função, o tipo mostrado pelo REPL é:

```
[< `Null | `Num of float | `String of bytes ] -> bytes
```

Isso significa que o parâmetro aceito pela função pode ser apenas uma das variantes `` `Null ``, `` `Num `` ou `` `String ``, mas nenhuma outra (pois nenhuma outra variante é prevista no `match`). O parâmetro também pode ser de um tipo que inclui menos variantes do que essas três, e por isso o caractere `<`. Por exemplo:

```
# type num_ou_null = [ `Null | `Num of float ];;
type num_ou_null = [ `Null | `Num of float ]

# let phi : num_ou_null = `Num ((1.0 +. sqrt 5.0) /. 2.0);;
val phi : num_ou_null = `Num 1.6180339887498949

# valor_string phi;;
- : bytes = "1.61803398875"
```

Nesse exemplo, inicialmente definimos o tipo `num_ou_null` que contém as duas variantes polimórficas `Num` e `Null`. Depois declaramos a variável `phi` do tipo `num_ou_null` (usando declaração explícita do tipo), e passamos essa variável para a função `valor_string`, o que funciona sem problemas, pois o tipo de `valor_string` começa com `<`.

Se fizermos um `match` com um padrão coringa, obtemos um tipo *aberto*. A função a seguir calcula para cada valor um índice que depende do tipo do valor:

```
# let valor_indice_tipo v =
    match v with
    | `Null -> 0
    | `Num _ -> 1
    | `String _ -> 2
    | _ -> 100;;
val valor_indice_tipo :
  [> `Null | `Num of 'a | `String of 'b ] -> int = <fun>
```

O tipo de `valor_indice_tipo` começa com `>`, representando o fato de que essa função agora aceita outras variantes polimórficas além das três especificadas:

```
# valor_indice_tipo @@ `Char 'a';;
- : int = 100
```

Assim como outros usos de padrões coringa, esse também pode ser perigoso, porque absolutamente qualquer variante polimórfica será aceita por essa função, mesmo variantes cuja intenção seja completamente diferente da de representar um valor JSON.

Variantes com valor

Nos exemplos anteriores, usamos variantes com um valor associado sempre com um valor de mesmo tipo, mas isso não é obrigatório. A flexibilidade das variantes polimórficas também permite o uso da mesma variante com valores de tipos diferentes:

```
# `Num 33.4;;
- : [> `Num of float ] = `Num 33.4

# `Num "pi";;
- : [> `Num of bytes ] = `Num "pi"
```

Mas essa flexibilidade também pode resultar em situações problemáticas. Uma forma fácil de causar problemas é usando funções com tipos que incluem variantes polimórficas com valores de tipos diferentes:

```
# valor_string;;
- : [< `Null | `Num of float | `String of bytes ] -> bytes =
  <fun>

# let valor_string2 v =
    match v with
    | `Null -> "(null)"
    | `Num s -> s
    | `String s -> s;;
val valor_string2 :
  [< `Null | `Num of bytes | `String of bytes ] -> bytes =
  <fun>

# let duas_strings v = (valor_string v) ^ (valor_string2 v);;
val duas_strings :
  [< `Null | `Num of bytes & float | `String of bytes ] ->
  bytes = <fun>
```

A função `valor_string` é a mesma definida antes. Definimos `valor_string2`, que recebe um valor com as mesmas variantes polimórficas, mas a variante `Num` tem valor `string`. Ao criar a função `duas_strings` chamando as duas funções, a variante `Num` aparece com um valor de tipo `bytes & float`.

Ou seja, para passar um valor com a variante `Num` para `duas_strings`, é preciso que ela carregue um valor que tenha tipo ao mesmo tempo `bytes` e `float`, o que é impossível. Por isso o erro a seguir:

```
# duas_strings (`Num 2.2);;
Error: This expression has type [> `Num of float ]
       but an expression was expected of type
         [< `Null | `Num of bytes & float | `String of bytes ]
       Types for tag `Num are incompatible
```

Mas as outras variantes funcionam normalmente com essa função:

```
# duas_strings `Null;;
- : bytes = "(null)(null)"
```

É recomendável utilizar os valores associados a variantes polimórficas de forma consistente. Caso se use uma variante com valores de tipos diferentes, deve-se evitar misturar esses usos. Para ter mais garantia, pode-se definir um tipo com as variantes polimórficas que podem ser usadas:

```
type valor = [ `Null | `Num of float | `String of string ]
```

O tipo `valor` especifica exatamente as variantes que fazem parte do tipo; qualquer função que aceite um valor do tipo `valor` como parâmetro deve aceitar exatamente essas três variantes, nem menos, nem mais. Os valores associados em cada variante também devem ser fixos. Mesmo assim, as variantes `` `Null ``, `` `Num `` e `` `String `` ainda podem ser usadas em outros tipos; declarar um tipo fixo apenas permite restringir as usadas em certos contextos.

Voltamos ao exemplo que motivou o uso de variantes polimórficas: a representação de valores dos formatos JSON e BSON.

Valores JSON e BSON

Com variantes polimórficas, podemos fazer os tipos JSON e BSON compartilharem as variantes de interesse, ao mesmo tempo que cada um inclui variantes que o outro não tem. Podemos definir os tipos em um mesmo módulo:

```
type json = [
| `String of string
| `Num of float
| `Null
| `Bool of bool
| `Lista of json list
| `Obj of (string * json) list ]

type bson = [
| `String of string
| `Null
| `Bool of bool
| `Lista of bson list
| `Obj of (string * bson) list
| `Data of int ]
```

Algumas das variantes são as mesmas nos dois tipos (por exemplo, `` `String ``), algumas só existem em um dos tipos (`` `Num `` e `` `Data ``), e outras existem nos dois tipos mas com valores de tipos diferentes (`` `Lista `` e `` `Obj ``, que são recursivos). Tudo isso funciona sem problemas com variantes polimórficas. Considerando que as funções que queremos escrever sempre processam apenas valores JSON, apenas valores BSON, ou fazem a conversão de JSON para BSON (ou vice-versa) em apenas uma direção, o uso das variantes com valores de tipos diferentes não deve causar problemas.

Embora o uso de variantes polimórficas elimine o problema do

compartilhamento de nomes das variantes, ainda pode ser desejável criar módulos separados para os valores JSON e BSON. Se fizermos isso usando variantes polimórficas, veremos uma característica interessante da interação entre variantes polimórficas e módulos:

```
module JSON = struct
  type t = [
  | `String of string
  | `Num of float
  | `Null
  | `Bool of bool
  | `Lista of t list
  | `Obj of (string * t) list ]
end

module BSON = struct
  type t = [
  | `String of string
  | `Null
  | `Bool of bool
  | `Lista of bson list
  | `Obj of (string * bson) list
  | `Data of int ]
end
```

Os dois módulos incluem as mesmas variantes polimórficas, mas como estas aparecem em módulos diferentes, elas são iguais ou diferentes? Ou seja, existe uma diferença entre a variante `` `Null `` no módulo `JSON` e a variante `` `Null `` no módulo BSON, como existe com as variantes tradicionais?

Se tentarmos acessar uma variante polimórfica em um módulo, vemos que algo está errado:

```
# JSON.`Null;;
Error: Syntax error
```

Analisando a situação, vemos que existe uma opção que faz mais sentido: variantes polimórficas nunca são declaradas, elas simplesmente existem. No nível conceitual, podemos pensar que todas as polimórficas possíveis existem o tempo todo, mas cada programa só usa algumas delas (obviamente a implementação não

segue essa ideia). Como elas não são declaradas, não faz sentido que pertençam a um módulo.

Variantes polimórficas são *globais*, e não seguem as mesmas regras de escopo dos outros valores da linguagem. Isso permite que mesmo tipos em módulos diferentes compartilhem variantes. A variante `` `Null `` do módulo `JSON` é a mesma do módulo `BSON`. Apesar das vantagens, isso também pode causar problemas, por exemplo, se dois módulos não relacionados em um programa grande (talvez dois módulos em bibliotecas diferentes usadas pelo programa) usarem, por acaso, uma mesma variante polimórfica, possibilitando que um mesmo valor possa ser usado em dois tipos sem relação.

Assim como acontece com os parâmetros rotulados, a decisão de usar variantes polimórficas deve pesar cuidadosamente as vantagens e desvantagens. As polimórficas podem ser encaradas como um sistema de tipagem dinâmica embutido na linguagem OCaml, com a diferença de que o programador deve incluir as *tags* de cada valor explicitamente, enquanto, em uma linguagem com tipagem dinâmica como Python, a *tag* de cada valor é inserida pelo interpretador.

Assim como a tipagem dinâmica, as variantes polimórficas trazem mais flexibilidade e comodidade em algumas situações, mas também apresentam armadilhas que podem tornar o programa menos confiável. Saber quando usá-las ou não fica mais fácil com a experiência de escrever e ler código na linguagem OCaml.

Se a motivação para usar variantes polimórficas é possibilitar a extensão de um tipo com novas variantes a qualquer momento, existe a possibilidade de usar as variantes extensíveis, como veremos a seguir.

11.3 VARIANTES EXTENSÍVEIS

As variantes extensíveis são uma novidade da versão 4.02 da linguagem, lançada durante a escrita deste livro. Elas permitem a criação de tipos abertos, para os quais novos construtores podem ser adicionados a qualquer momento. Isso é similar ao comportamento das exceções que já existiam na linguagem, mas agora esse comportamento está disponível para tipos criados pelo usuário.

Tipos variantes extensíveis são definidos usando a seguinte sintaxe:

```
type valor = ..
```

O tipo `valor` é declarado como um tipo extensível usando os dois pontos `..` como definição do tipo. A partir daí, o tipo `valor` pode ser estendido com novas variantes a qualquer momento:

```
type valor += String of string
```

Também podemos adicionar várias novas variantes de uma vez:

```
type valor += Int of int | Float of float
```

Para fazer `match` com um valor de um tipo variante extensível, é necessário incluir um padrão coringa (já que é impossível antecipar quais variantes serão passadas como parâmetro):

```
let valor_string v =
  match v with
  | String s -> s
  | Int i -> Printf.sprintf "%d" i
  | Float f -> Printf.sprintf "%f" f
  | _ -> "???"
```

Se o padrão coringa não for incluído, o compilador emite um aviso de `match` não exaustivo, mas o código é compilado. Entretanto, qualquer chamada que use uma variante que não é uma das três já definidas causará uma exceção `Match_failure`, portanto é recomendável incluir o padrão coringa.

Como discutimos no capítulo *Registros e variantes*, usar o padrão coringa significa que o compilador não vai mais verificar se o `match` é exaustivo, já que este aceita qualquer valor. No caso das variantes extensíveis, isso é necessário, o que significa que o uso dessa característica, mais uma vez, resulta em um aumento de conveniência para o programador, mas a um preço. Nesse caso, o preço é uma redução na confiabilidade do código, já que o compilador tem uma possibilidade a menos para detectar erros.

Exemplo de uso dos tipos extensíveis

Porém, existem situações nas quais compensa pagar o preço. Uma das coisas que se ganha com tipos extensíveis é a possibilidade do usuário de uma biblioteca poder definir novas variantes para algum tipo da biblioteca, e dessa forma usar *pattern matching* para situações em que não seria possível com os tipos variantes tradicionais.

Um exemplo é uma biblioteca para mensagens. Vamos imaginar que estamos criando uma biblioteca que se encarrega do transporte de mensagens sobre vários suportes e protocolos (algo similar a ZeroMQ). Nossa biblioteca vai se certificar de que o transporte das mensagens é feito de maneira confiável, mas gostaríamos de dar aos usuários da biblioteca a flexibilidade de definir suas próprias mensagens.

Uma forma de fazer isso é definir um tipo genérico para mensagens, e usar um campo desse tipo para identificar a mensagem específica. Se estivéssemos trabalhando na linguagem C, poderíamos pensar em um tipo como o seguinte:

```
type msg = {
  id : int;
  cont : bytes
}
```

O campo `id` guarda o identificador da mensagem como um número inteiro, enquanto o campo `cont` é o conteúdo da mensagem, na forma de um *buffer* arbitrário de `bytes`. A aplicação que usa a biblioteca deve se encarregar de definir e usar os identificadores de maneira consistente, e de serializar e desserializar o conteúdo da mensagem de maneira correta de acordo com seu tipo. É um estilo de programação de baixo nível.

Uma possibilidade mais interessante seria definir um tipo polimórfico para mensagens, de acordo com o conteúdo:

```
type 'a msg = {
  id : int;
  cont: 'a
}
```

Uma aplicação definiria o tipo dos conteúdos, de acordo com os tipos de mensagem necessários, por exemplo:

```
type conteudo = String of string | Arq of string * string
```

A partir daí, a aplicação usaria mensagens de tipo `conteudo msg`. A biblioteca precisaria ter a capacidade de serializar mensagens do tipo polimórfico, para qualquer tipo. Isso pode não ser simples de fazer sem alguma ajuda do usuário da biblioteca, mas é possível (talvez usando um pouco de metaprogramação).

Uma forma similar, mas mais clara, seria usando tipos variantes extensíveis. O tipo para mensagens incluído na biblioteca seria extensível:

```
type msg = ..
```

A aplicação definiria as variantes que precisasse, uma para cada mensagem. O conteúdo de cada mensagem seria representado pelos valores associados às variantes:

```
type msg +=
| Parar
```

```
| AbrirArq of string * string
| Dormir of int
```

A biblioteca também teria de saber como serializar e transmitir mensagens de variantes arbitrárias, porém agora a aplicação poderia processar mensagens usando `match`:

```
let processar_msg m =
  match m with
  | Parar -> parar_aplicacao ()
  | AbrirArq (dir, nome) -> abrir_arquivo dir nome
  | Dormir segs -> dormir segs
  | _ -> failwith "Tipo de mensagem inesperado"
```

Cada aplicação poderia definir apenas as variantes que tivesse interesse, usá-las como se o tipo fosse definido pela própria aplicação, mas aproveitar toda a infraestrutura de transporte de mensagens provida pela biblioteca. Nesse caso, o uso de tipos extensíveis é uma solução elegante para o problema.

11.4 VARIANTES DE TIPOS VARIANTES

Neste capítulo, vimos duas variações dos tipos variantes tradicionais que foram introduzidos no capítulo *Registros e variantes* e usados ao longo do livro: variantes polimórficas e tipos variantes extensíveis. Ambas adicionam novas capacidades aos tipos variantes, ao preço de maior complexidade e/ou menor confiabilidade.

Uma questão importante que deve ser considerada quando se pensa em usar essas variações em uma situação é se os tipos variantes tradicionais não são suficientes. Algumas vezes, programadores com pouca experiência na linguagem decidem usar variantes polimórficas em situações em que as tradicionais seriam perfeitamente suficientes.

Além de o desempenho das variantes polimórficas ser um pouco

inferior, usá-las pode facilmente resultar em código mais frágil, que apresenta erros ou comportamento estranho. Quando se decidir usar as variantes polimórficas, é preferível definir tipos fixos que incluem apenas as variantes necessárias, sempre que possível.

Os tipos variantes extensíveis não foram muito utilizados ainda, por serem uma adição recente à linguagem, mas as mesmas observações valem. Os tipos variantes extensíveis são menos flexíveis que as polimórficas, e devem ter desempenho similar aos tipos variantes tradicionais. Porém, a capacidade de definir novas variantes a qualquer momento também pode reduzir a confiabilidade do código. Por isso, os tipos variantes extensíveis devem ser usados apenas quando os tipos variantes tradicionais não são suficientes.

Uma outra extensão dos tipos variantes tradicionais são os GADTs (*Generalized Algebraic Data Types*), também adicionados recentemente à linguagem (na versão 4.0). GADTs permitem construtores cujo tipo de resultado é arbitrário, possibilitando representar um conjunto maior de informações no próprio tipo.

Não vamos falar sobre GADTs aqui por serem uma característica mais avançada e recente da linguagem, para a qual ainda não existe um consenso sobre as boas práticas de uso. Quem tiver mais curiosidade pode consultar a seção do manual de referência sobre GADTs, em http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec238.

Já que falamos sobre JSON e BSON, existem algumas bibliotecas já prontas para trabalhar com esses formatos na linguagem OCaml.

Para JSON, uma boa biblioteca é a Yojson: http://mjambon.com/yojson.html.

Para o formato BSON, existe Bson.ml:

http://massd.github.io/bson/.

Capítulo 12

UM POUCO SOBRE OBJETOS

Até esse ponto do livro, temos ignorado que o O do nome da linguagem OCaml originalmente significava *Objective*, quando o nome oficial da linguagem ainda era *Objective Caml*. Houve uma época em que o paradigma OO estava na moda, por assim dizer, e muitas linguagens se apressaram em adicionar características OO para entrarem na tendência (assim como hoje muitas linguagens adquirem características funcionais para serem mais *cool*).

OCaml também seguiu essa tendência ao incluir características OO à linguagem antes conhecida como Caml light. Mas, diferente do que foi feito em outras, com OCaml isso foi feito do jeito que se espera que aconteça para uma linguagem criada por um instituto de pesquisa e inovação: através de um longo projeto de pesquisa que buscou a melhor forma de integrar características OO a uma linguagem funcional estaticamente tipada com um sistema de tipos Hindley-Milner.

A interação entre o mecanismo de herança, historicamente considerado importante no paradigma OO (mas hoje malvisto mesmo entre os defensores do paradigma), e a inferência de tipos global do sistema Hindley-Milner sempre foi considerada problemática. O sistema de objetos em OCaml resolve os problemas mais sérios, mas ainda tem algumas desvantagens. O uso de objetos em OCaml dificulta a inferência de tipos e quase sempre requer

mais declarações explícitas de tipo.

Por esse e outros motivos, o sistema de objetos da linguagem não é muito popular na comunidade. É bastante interessante e diferente do usual, mas o que se consegue fazer com objetos pode ser feito com outras características da linguagem e, dessa forma, a maior parte da comunidade prefere não lidar com objetos.

Mesmo assim, as características OO da linguagem OCaml são usadas em algumas situações e bibliotecas importantes, incluindo as bibliotecas de interface gráfica mais maduras e bem mantidas. Por isso, é interessante para um programador OCaml conhecer pelo menos o básico sobre o sistema de objetos da linguagem, e esse é o objetivo deste capítulo. É uma visão rápida para saber pelo menos qual é a aparência dos objetos, classes e métodos na linguagem.

12.1 OBJETOS

Objetos em OCaml podem ser criados sem que seja criada uma classe antes. É possível definir diretamente os componentes de um objeto usando uma *expressão de objeto*, começando com a palavra-chave `object`:

```
# let p =
    object
      val mutable x = 0
      method get_x = x
      method inc_x = x <- x + 1
    end;;
val p : < get_x : int; inc_x : unit > = <obj>
```

O objeto `p` tem um campo, `x`, que é mutável, e dois métodos: `get_x` e `inc_x`. O primeiro retorna o valor do campo `x`, e o segundo incrementa o valor de `x`.

A resposta do REPL mostra que `p` é um objeto com uma interface composta por esses dois métodos, e inclui também o tipo

de retorno de cada um. O compilador infere que o campo `x` tem tipo `int` pelo valor de inicialização, que é uma constante inteira; portanto, `get_x` tem tipo de retorno `int`.

O método `inc_x` realiza uma mudança de valor em um campo mutável, similar ao mecanismo para registros, e isso é um efeito imperativo que tem resultado `()`; logo, o método `inc_x` tem tipo de retorno `unit`. Os dois métodos não recebem parâmetros; o objeto receptor (conhecido em muitas linguagens como `self` ou `this`) é implícito na chamada e no código dos métodos.

A sintaxe para chamada de métodos é `obj#met`, em que `obj` é o objeto e `met` é o nome do método:

```
# p#get_x;;
- : int = 0

# p#inc_x;;
- : unit = ()

# p#get_x;;
- : int = 1
```

Funções que usam objetos conseguem inferir parte da interface deles, e objetos que satisfazem a parte inferida podem ser passados como parâmetro. Vamos definir uma função que chama o método `get_x` no valor passado como parâmetro e multiplica por dois:

```
# let get_x_dobro o =
    o#get_x * 2;;
val get_x_dobro : < get_x : int; .. > -> int = <fun>
```

O tipo inferido para a função `get_x_dobro` é interessante: `< get_x : int; .. >` é o tipo de um objeto que tem um método `get_x` retornando um inteiro, e possivelmente outros métodos quaisquer (indicados pelos caracteres `..`). Como o objeto `p` criado antes satisfaz essa interface, podemos passá-lo para essa função:

```
# get_x_dobro p;;
```

```
- : int = 2
```

Uma vez que criamos um objeto como `p`, podemos querer criar outras instâncias com a mesma interface e mesmos campos. Ou podemos querer derivar objetos a partir de outros já existentes, como acontece na programação OO. Para isso, precisamos não só criar objetos isolados, mas também classes para servir como modelos para a criação de novos objetos.

12.2 CLASSES

As classes funcionam como um modelo a partir do qual vários objetos que seguem o modelo podem ser criados. Em OCaml, uma classe também é criada usando uma expressão objeto, mas com a declaração `class` e o nome da classe antes:

```
class contador =
  object
    val mutable c = 0
    method get = c
    method inc = c <- c + 1
    method dec = c <- c - 1
    method inc_n n = c <- c + n
  end
```

O compilador infere o tipo dos métodos assim como o esperado:

```
class contador :
  object
    val mutable c : int
    method dec : unit
    method get : int
    method inc : unit
    method inc_n : int -> unit
  end
```

O tipo da classe `contador` descreve todos os componentes da classe. Um objeto da classe pode ser criado com o operador `new`:

```
# let c1 = new contador;;
val c1 : contador = <obj>
```

O tipo do objeto `c1` é a sua classe, `contador`. Com a classe, podemos criar vários objetos similares, cada um com um estado separado:

```
# c1#inc_n 10;;
- : unit = ()

# let c2 = new contador;;
val c2 : contador = <obj>

# c2#inc_n 2;;
- : unit = ()

# c1#get;;
- : int = 10

# c2#get;;
- : int = 2
```

Uma classe em OCaml pode ser encarada como uma função que cria objetos seguindo um certo modelo. Para criar objetos passando um parâmetro de construção, deixamos esse fato mais explícito:

```
class contador = fun init ->
  object
    val mutable c = init
    method get = c
    method inc_n n = c <- c + n
  end
```

Assim como nas definições de função usando `let`, também podemos usar uma forma mais compacta de definir uma classe. O código a seguir é equivalente ao exemplo anterior:

```
class contador init =
  object
    val mutable c = init
    method get = c
    method inc_n n = c <- c + n
  end
```

O parâmetro deve ser passado na hora de criar o objeto usando `new`:

```
# let c1 = new contador 10;;
val c1 : contador = <obj>

# c1#get;;
- : int = 10
```

Note que usar `new` na nova classe `contador` sem passar o parâmetro resulta em algo similar a uma aplicação parcial de função. O resultado é uma função, que pode ser depois aplicada ao parâmetro, gerando um objeto:

```
# let c2 = new contador;;
val c2 : int -> contador = <fun>

# let c3 = c2 42;;
val c3 : contador = <obj>

# c3#get;;
- : int = 42
```

Nesse exemplo, não existe vantagem em fazer uma aplicação parcial, mas podemos ver que as regras de aplicação parcial que funcionam com funções também valem para a criação de instâncias (objetos) de uma classe. Isso nos permite criar novos construtores pela aplicação parcial da criação de um objeto de uma classe.

O sistema de objetos em OCaml é bastante completo e inclui mais características interessantes: interfaces, métodos virtuais, herança, classes parametrizadas (que funcionam similarmente aos genéricos de outras linguagens OO), entre outros detalhes.

Muito do que se faz com objetos em linguagens OO é feito com módulos e funtores em OCaml, principalmente agora que existem módulos de primeira classe na linguagem. Por isso, poucos programadores da comunidade usam o O da linguagem OCaml. Mas existem situações em que as classes podem ser uma melhor solução do que usar módulos e funtores. Por isso, pode valer a pena eventualmente estudar o sistema de objetos mais a fundo.

Aqui não passaremos dessa rápida introdução, mas os

interessados podem ler mais sobre o assunto no tutorial que consta no manual de referência da linguagem: http://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html

O livro Real World OCaml também contém dois capítulos sobre o sistema de objetos da linguagem: o capítulo 11, *Objects*, em http://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html; e o capítulo 12, *Classes*, em https://realworldocaml.org/v1/en/html/classes.html.

Capítulo 13

ORGANIZAÇÃO DE PROJETOS E TESTES

O dia a dia do uso de uma linguagem de programação em projetos reais vai muito além apenas da linguagem em si. Embora o foco deste livro e a maior parte dele tenham se concentrado nas características de OCaml e no paradigma de programação funcional, neste capítulo vamos mudar a direção para mostrar algumas ferramentas que podem ser usadas para facilitar o desenvolvimento de projetos usando OCaml.

Vamos examinar aqui o programa OASIS para especificação de projetos e criação de sistemas de *build*, e a biblioteca OUnit para testes de unidade. Nosso estudo de caso será um projeto simples: o conjunto de interpretador, compilador e máquina virtual para expressões aritméticas do capítulo *Exemplo: interpretador e compilador*.

13.1 ORGANIZAÇÃO DO PROJETO COM OASIS

OASIS é uma ferramenta para descrição de projetos usando a linguagem OCaml. Através de uma descrição declarativa, o OASIS cria um sistema de *build* para o projeto, e também pode gerar um pacote para distribuí-lo na plataforma OPAM.

OASIS usa a *findlib* para gerenciar dependências e *ocamlbuild* para o sistema de *build*. Isso normalmente é suficiente para projetos

de pequeno e médio porte. Porém, usuários com projetos de grande porte podem preferir usar outra ferramenta de *build*.

Para o exemplo a seguir, é interessante ter acesso ao código-fonte dos exemplos do livro e olhar os arquivos discutidos no diretório `exp`.

A descrição de um projeto deve ser colocada em um arquivo de nome `_oasis`, geralmente no diretório principal do projeto. A descrição para o projeto da linguagem de expressões no OASIS é bastante simples, começando com um cabeçalho:

```
OASISFormat:  0.4
Name:         Exp
Version:      0.0
Synopsis:     Exemplo de compilacao e interpretacao
Authors:      Andrei Formiga
License:      MIT
Plugins:      META (0.4)
BuildTools:   ocamlbuild
```

Isso define o nome do projeto (`Exp`), a versão, autor, uma descrição, a licença do código, o sistema de *build* utilizado, e os *plugins* necessários. O *plugin* META faz com que o OASIS gere um arquivo META para o projeto; esse arquivo é necessário para registrar uma biblioteca na findlib.

Outro *plugin* frequentemente usado é o DevFiles, que faz o OASIS gerar um `Makefile` e um script de configuração para compilação com a sequência `configure/make/make install`, comum em projetos em sistemas Unix.

Após o cabeçalho, devem aparecer seções especificando os artefatos que fazem parte do projeto (os principais são biblioteca, executável, testes e documentação). No caso do projeto `Exp`, temos uma biblioteca que define as funções vistas no capítulo *Exemplo: interpretador e compilador*.

```
Library exp
```

```
  Path:           src
  Modules:        Exp
  BuildDepends:
  CompiledObject: best
  NativeOpt:      -inline 20
```

O principal desta seção é que é definida uma biblioteca `exp`, cujo código está no diretório `src` (relativo ao diretório do projeto), e que os módulos que fazem parte desta biblioteca são apenas o módulo `Exp` (compilado a partir do arquivo `exp.ml`). As demais declarações são opcionais, na maioria opções de compilação.

Em seguida, declaramos mais duas seções: um executável que inclui os testes de unidade do projeto, e a especificação de que esse executável é um programa de teste para o projeto:

```
Executable test
  Path:           test
  MainIs:         test.ml
  Install:        false
  Build$:         flag(tests)
  BuildDepends:   oUnit (>= 2.0.0), exp
  CompiledObject: best
  ByteOpt:
  NativeOpt:      -inline 20

Test main
  Command:        $test
```

Essas especificações dizem que o programa de teste está no arquivo `test.ml` no diretório `test`, que esse executável só deve ser compilado se a opção de compilação (*flag*) `tests` estiver ativado, e que o teste depende da biblioteca OUnit para testes de unidade e da biblioteca `exp` declarada antes.

Isso é o suficiente para descrever o projeto. Com o OASIS instalado, basta executar o seguinte comando no diretório do projeto (o diretório que contém o arquivo `_oasis`):

```
$ oasis setup
```

Isso gera um arquivo `setup.ml` que contém o programa

principal do sistema de *build*, assim como outros arquivos de suporte. Para compilar o programa, deve-se executar o `setup.ml` passando as opções adequadas:

```
ocaml setup.ml -configure
ocaml setup.ml -build
```

Para habilitar e executar os testes, pode-se usar a seguinte sequência:

```
ocaml setup.ml -configure --enable-tests
ocaml setup.ml -test
```

Uma vez gerados o arquivo `setup.ml` e outros auxiliares, os usuários do projeto não precisam usar OASIS. O sistema de *build* criado (que utiliza ocamlbuild) é autocontido, e apenas desenvolvedores de projetos precisam ter o OASIS instalado. Caso o plugin DevFiles seja usado, o projeto pode ser compilado com a conhecida sequência:

```
./configure
make
```

A configuração pode habilitar a compilação de testes com o uso da opção `--enable-tests`.

Isso é o suficiente para descrever o projeto e criar um sistema de *build* para ele que leva em consideração as dependências de outras bibliotecas e que funciona em todas as plataformas em que OCaml é suportado (incluindo Windows, através do arquivo `setup.ml`). A ferramenta `oasis2opam` permite a criação de um pacote para a plataforma OPAM através da descrição do projeto seguindo o padrão do OASIS. Essa ferramenta não está incluída no OASIS e deve ser instalada separadamente.

A forma mais comum de usar o OASIS é copiando descrições de projetos similares e adaptando a descrição para suas necessidades. Para mais detalhes sobre o OASIS, o site oficial da ferramenta é

http://oasis.forge.ocamlcore.org/.

13.2 TESTES COM OUNIT

Vimos no capítulo *Exemplo: interpretador e compilador* que o interpretador para a linguagem de expressões pode ser escrito em poucas linhas:

```
let rec eval e =
  match e with
    Const n -> n
  | Adic (e1, e2) -> eval e1 + eval e2
  | Sub (e1, e2) -> eval e1 - eval e2
  | Mult (e1, e2) -> eval e1 * eval e2
```

Mesmo o interpretador sendo simples a ponto de parecer "obviamente correto", é importante sempre testar o código que escrevemos. Além de aumentar a confiabilidade do código e ajudar a detectar defeitos, ter testes automatizados para o código nos dá suporte para várias outras atividades de desenvolvimento, como a refatoração do código. Os testes criados para o código do capítulo *Exemplo: interpretador e compilador* detectaram um *bug* na função principal do compilador.

Existem várias formas de testar código, incluindo algumas que foram criadas na comunidade da programação funcional. Uma forma bastante conhecida são os testes de unidade, principalmente seguindo as ideias da biblioteca de testes de unidade JUnit, para a linguagem Java. OCaml possui uma biblioteca de testes nesses moldes, que se chama, previsivelmente, OUnit.

OUnit funciona como a maioria das bibliotecas similares para testes de unidade, mas possui algumas características adicionais que são interessantes. Por exemplo, como testes em OUnit são funções, e funções em OCaml são cidadãos de primeira classe, temos testes de primeira classe em OUnit. Isso possibilita vários tipos de

manipulação dos testes para criar alguns tipos de baterias de testes de forma bem mais simples do que é possível com JUnit e similares. Aqui veremos apenas uma introdução ao OUnit, sem explorar suas características mais avançadas.

Para usar OUnit, a biblioteca deve estar instalada (o que é fácil de fazer usando OPAM) e o projeto deve estar configurado para incluí-la na compilação. Uma sugestão simples é organizar o sistema de *build* do projeto usando OASIS, como vimos anteriormente.

Se tudo estiver configurado corretamente, usar OUnit para testar seu código é simples. Normalmente cria-se um arquivo separado para testes. No começo deste arquivo, é necessário incluir as linhas:

```
open OUnit2
open Exp
```

Isso abre o módulo principal da biblioteca e do programa que estamos testando. Embora tenhamos visto que abrir módulos não seja aconselhável em todas as situações, nesse caso o código dos testes fica muito mais conciso com essas aberturas.

O arquivo com o programa de testes se chama `test.ml`, no diretório `test`. Neste arquivo, vamos escrever um teste para a função do interpretador, `eval`. Ele consiste em interpretar algumas expressões de exemplo e verificar se o valor calculado pelo interpretador confere com o esperado. As expressões usadas no teste são:

```
(* 4 + 3 * 2 *)
let e1 = Soma (Const 4, Mult (Const 3, Const 2))

(* (4 + 3) * 2 *)
let e2 = Mult (Soma (Const 4, Const 3), Const 2)

(* (4 + 3) * 2 + 5 *)
let e3 = Soma (Mult (Soma (Const 4, Const 3), Const 2), Const 5)
```

```
(* (11 + 0) * (9 - 2) *)
let e4 = Mult (Soma (Const 11, Const 0), Sub (Const 9, Const 2))
```

Agora, vamos usar essas expressões para criar a função de teste. Uma função de teste para a biblioteca OUnit2 deve receber um parâmetro: o contexto de execução do teste. Como não precisamos manipular esse contexto para testar a função `eval`, vamos ignorá-lo. A função de teste é:

```
let t_eval ctxt =
  let pares = [(e1, 10); (e2, 14); (e3, 19); (e4, 77)] in
  List.iter (fun (e, v) -> assert_equal (eval e) v) pares
```

A lista `pares` contém pares nos quais o primeiro componente é uma expressão, e o segundo é o resultado esperado para a expressão. A função mais comum para testes em OUnit é `assert_equal`. Ela recebe dois parâmetros principais: o valor calculado pelo programa e o valor esperado.

`assert_equal` tem tipo de retorno `unit`, portanto, não retorna nada. Se os valores forem diferentes, uma exceção é disparada, e se isso ocorre em um teste, a biblioteca captura a exceção e conta como uma falha de teste. A função de teste também deve ter tipo de retorno `unit`; normalmente, ela será apenas uma sequência de asserções. A função de teste `t_eval` é uma sequência de `assert_equal`, organizada com uma chamada a `List.iter`. A mesma função poderia ser escrita da seguinte forma:

```
let t_eval ctxt =
  assert_equal (eval e1) 10;
  assert_equal (eval e2) 14;
  assert_equal (eval e3) 19;
  assert_equal (eval e4) 77
```

De uma forma ou de outra, temos um teste, e é necessário incluí-lo em alguma bateria (*suite*) de testes para poder executá-lo. A bateria é criada usando um operador específico que associa um nome a uma lista de testes; cada teste na lista que faz parte da bateria também é associado a um nome usando outro operador:

```
let suite =
  "bateria01" >:::
    [ "interpretador" >:: t_eval ]
```

Note que o operador que associa o nome da bateria à lista de testes usa três caracteres dois-pontos, e o operador que associa o nome de cada teste à sua função usa apenas dois caracteres dois-pontos.

Com uma bateria de testes, podemos usar uma função executora do OUnit para rodar todos os testes da bateria. Uma escolha comum de função executora é `run_test_tt_main`, e podemos criar uma função principal para o arquivo `test.ml` que apenas roda o executor dos testes:

```
let () =
  run_test_tt_main suite
```

Agora, se rodarmos o teste, obtemos a saída:

```
andrei$ ocaml setup.ml -test
.
Ran: 1 tests in: 0.08 seconds.
OK
```

Isso mostra que o único teste que temos foi executado e bem-sucedido.

Testando a máquina de pilha

Para a máquina de pilha, criamos alguns programas de teste:

```
let p1 = [Empilha 5; Empilha 3; Oper OpSoma]

let p2 = [Empilha 2; Empilha 9; Oper OpSub;
          Empilha 0; Empilha 11; Oper OpSoma; Oper OpMult]
```

E criamos uma função para testar o executor da máquina de pilha:

```
let t_exec ctxt =
  let pares = [(p1, Some 8); (p2, Some 77)] in
```

```
  List.iter (fun (p, v) -> assert_equal (executa p) v) pares
```

O programa `p2` implementa a expressão `e4` vista antes. É claro que, para testes mais realistas, teríamos de incluir mais casos, por exemplo programas que não tenham valor porque incluem uma sequência de instruções inválida, como chamar uma operação sem ter valores na pilha.

Esse teste é incluído na suíte:

```
let suite =
  "bateria01" >:::
    [
      "interpretador" >:: t_eval;
      "maquina de pilha" >:: t_exec
    ]
```

E executar o teste agora mostra dois testes bem-sucedidos.

Testando o compilador

Da mesma forma, criamos dois testes envolvendo o compilador: um que verifica se a expressão `e4`, quando compilada, é igual ao programa `p2` (poderíamos incluir outros pares de teste), e outro que verifica se uma expressão compilada e executada na máquina de pilha chega no resultado esperado. O código dos testes é:

```
let t_compila ctxt =
  assert_equal (compila e4) p2

let t_compila_executa ctxt =
  let comp_exec e =
    e |> compila |> executa |> valor_option
  in
  let pares = [(e1, 10); (e2, 14); (e3, 19); (e4, 77)] in
  List.iter (fun (e, v) -> assert_equal (comp_exec e) v) pares
```

Incluímos esses testes na bateria:

```
let suite =
  "bateria01" >:::
    [
      "interpretador" >:: t_eval;
```

```
    "maquina de pilha" >:: t_exec;
    "compilador" >:: t_compila;
    "compilacao e execucao do programa" >:: t_compila_executa
  ]
```

A execução dos testes deve mostrar quatro sucessos. Em uma primeira versão do código, o compilador gerava código que empilhava os operandos na ordem inversa do que devem ser empilhados. Por exemplo, a expressão `9 - 2` gerava o código `[Empilha 9; Empilha 2; Oper OpSub]`. Isso não é problema para as operações de soma e multiplicação, que são comutativas. Mas quando o teste incluiu uma expressão com subtração, os dois testes que envolvem compilação falharam. Ou seja, mesmo uma bateria pequena, com pouca cobertura como essa, foi útil para revelar um bug importante no código.

Existe uma forma ainda mais simples de especificar testes usando a extensão sintática `pa_test`. Com ela, é possível criar um teste simplesmente usando a forma `TEST e`, em que `e` é a expressão de teste. A extensão se encarrega de reunir os testes em uma bateria e chamar o executor. Mas é interessante ver como usar a biblioteca OUnit diretamente.

Este capítulo foi uma introdução rápida às ferramentas OASIS e OUnit, que são bastante úteis no desenvolvimento de projetos na linguagem OCaml. Mais detalhes sobre OUnit podem ser encontrados no site da biblioteca: http://ounit.forge.ocamlcore.org/.